FORSCHUNGSZENTRUM JÜLICH GmbH
Zentralinstitut für Angewandte Mathematik
D-52425 Jülich, Tel. (02461) 61-6402

Interner Bericht

Lösung nichtlinearer Ausgleichsprobleme mit symbolisch-numerischen Rechenverfahren in Maple

René Külheim

FZJ-ZAM-IB-2003-07

Mai 2003

(letzte Änderung: 20.5.2003)

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Lösung nichtlinearer Ausgleichsprobleme mit symbolisch-numerischen Rechenverfahren in Maple

Modellierung von Daten, insbesondere mit nichtlinearen Modellfunktionen, ist in vielen Bereichen von Wissenschaft und Technik eine regelmäßig wiederkehrende Aufgabe. Ein Kennzeichen dieser Art der Augleichsrechnung ist, dass Lösungen nur durch Iterationsverfahren bestimmt werden können. In der Vergangenheit wurden Bibliotheksroutinen, wie etwa aus den NAG-Bibliotheken oder den Numerical Recipes verwendet, um Lösungen zu berechnen. Hierzu waren Kenntnisse der jeweiligen Programmiersprachen nötig. Zudem war die Implementierung von benötigten ersten und eventuell zweiten Ableitungen schwierig und fehleranfällig. Heutzutage werden für mathematische Modellierungen integrierte mathematische Softwaresysteme, wie z.B. Maple eingesetzt. In seinem derzeitigen Release verfügt das Maple-System jedoch über keine Funktionalität zum Lösen solcher Probleme. Im Rahmen dieser Diplomarbeit entstand ein Programmpaket, das zur Lösung nichtlinearer Ausgleichsprobleme symbolische und numerische Methoden kombiniert. Zusätzlich wurde eine graphische Oberfläche entwickelt, mit der Lösungen auf einfache und intuitive Weise berechnet werden können. In diesem Programmpaket werden zwei Verfahren zur numerischen Lösung angeboten: Der Levenberg-Marquardt-Algorithmus und ein modifizierter Gauß-Newton-Algorithmus. Benötigte partielle Ableitungen werden dabei mittels symbolischer Differentiation vom Maple-System berechnet.

Solving nonlinear fitting problems with symbolic-numeric methods in Maple

The modeling of data is a regular task in many areas of science and engineering. If the model function is nonlinear, a solution cannot be computed directly. An iterative method must be chosen instead. In the past, routines from NAG libraries or the Numerical Recipes were used to calculate the solution. In addition, the knowledge of a specific programming language was needed to use these routines. Furthermore, the implementation of required first and second order derivatives was difficult and error-prone. Nowadays, integrated mathematical software systems like Maple are used for mathematical modeling. The current Maple version has no functionality to solve such problems.

In the context of a diploma thesis a program package has been developed for solving nonlinear fitting problems by combining symbolic and numerical methods. In addition, a graphical user interface to this package enables an easy and intuitive solution of nonlinear fitting problems. Currently, two numerical methods are implemented: The Levenberg-Marquardt-Algorithm and a modified Gauss-Newton-Algorithm. The partial derivatives are calculated by the Maple system using symbolic differentiation methods.

Kapitel 1

Einleitung

Modellierung von Daten, insbesondere mit nichtlinearen Modellfunktionen, ist nicht nur im Forschungszentrum Jülich eine immer wieder auftretende Aufgabenstellung in Wissenschaft und Technik. Während bei der linearen Ausgleichsrechnung in einem Schritt eine eindeutige Lösung bestimmt werden kann, muss bei der nichtlinearen ein Iterationsverfahren angewandt werden, das nicht notwendigerweise zur optimalen Lösung führt. In Kapitel 2 werden Verfahren zum Lösen solcher Probleme näher beschrieben. Dabei werden zwei Algorithmen besonders hervorgehoben: Ein modifizierter Gauß-Newton-Algorithmus und der Levenberg-Marquardt-Algorithmus. Beide Verfahren werden in Kapitel 3 einem intensiven Test auf Robustheit und Effizienz anhand mehrerer repräsentativer Datensätze unterzogen. Bei der Anwendung der Algorithmen ergibt sich das Problem, dass Ableitungen erster und eventuell zweiter Ordnung zu bestimmen sind. Andererseits müssen Startwerte für die Parameter vorgegeben werden. Eine graphische und numerische Kontrolle der Güte der Anpassung ist im nichtlinearen Fall besonders wichtig.
In der Vergangenheit wurden zur Bestimmung nichtlinearer Modellfunktionen Bibliotheksroutinen eingesetzt, wie die aus den NAG-Bibliotheken (z.B. [18]) oder den Numerical Recipes ([5]). Eine visuelle Kontrolle der Ergebnisse erfordert die Anbindung von Graphikpaketen. Zunehmend werden für die mathematische Modellbildung integrierte interaktive Softwaresysteme mit flexiblen und problemorientierten Programmiermöglichkeiten und umfangreichen graphischen Ausgabemöglichkeit eingesetzt, wie z.B. Maple ([13]). Maple war ursprünglich ein reines Computeralgebrasystem, mit dem z.B. symbolische Ableitungen berechnet werden können. Inzwischen bietet das System umfangreiche Programmiermöglichkeiten mit Prozeduren und Modulen, von denen in Kapitel 4 die wichtigsten Merkmale beschrieben werden. Außerdem wird im Kapitel 4 die Möglichkeit beschrieben, externe Routinen in C und Fortran, insbesondere jedoch Bibliotheksfunktionen, in das bestehende Maple-System zu integrieren. Ab dem Release 8 des Maple Systems steht ein GUI[1] Tool zur Verfügung, um Anwendungen in Maple mit einer graphischen Oberfläche zu versehen. Diese sogenannten Maplets ermöglichen es dem Endbenutzer, interaktiv auf einfache und intuitive Weise ohne die Kenntnis der Maple-Kommandos mit dem System zu kommunizieren. Einen Überblick über die wesentlichen Merkmale der Maplets wird ebenfalls in diesem Kapitel gegeben.
Wie in Kapitel 5 näher erläutert wird, ist die Benutzung numerischer Ableitungen problematisch, die Angabe exakter Ableitungen eventuell aufwändig und bei ihrer Codierung fehleranfällig. Die automatische Differentiation bietet hier Abhilfe. Mit ihr können Ableitungen nicht nur von Funktionen, sondern auch von komplexen Programmen effizient berechnet werden. Kapitel 5 stellt wichtige Eigenschaften dieser Technik des Differenzierens vor.
Die Benutzung von Maple stellt im Vergleich zu reiner Fortran- oder C-Programmierung eine wesentliche Vereinfachung dar. Jedoch verfügt Maple in seinem derzeitigen Release über keine Funktionalität zum Lösen von nichtlinearen Ausgleichsproblemen. Ein erster Schritt wurde in [15] gemacht, wo ein Algorithmus in Maple-Quellcode zu deren Lösung angegeben wird. Im Zuge dieser

[1] GUI - Graphical User Interface

Diplomarbeit entstand daher ein Paket, das insbesondere die in Kapitel 4 vorgestellten Features nutzt. Die Einbindung von Bibliotheksfunktionen ist aus zwei Grüden sinnvoll: Einerseits kann man auf robuste und gut getestete Software zurückgreifen, andererseits ist die Implementierung der Verfahren mittels Maple-Programmierung weniger effizient, auch wenn in Maple Rechnungen in Hardware-Arithmetik möglich sind. Mit dem in Kapitel 6 vorgestellten Programmpaket können auf einfache Art und Weise Lösungen von nichtlinearen, mehrdimensionalen Ausgleichsrechnungen bestimmt werden. Neben den gewichteten nichtlinearen Fits ohne Nebenbedingungen können mit diesem Paket ebenfalls Interpolationen und lineare Fits durchgeführt werden. Um die Güte der Ergebnisse beurteilen zu können, stehen einerseits verschiedene graphische Ausgabemöglichkeiten zur Verfügung, andererseits kann die Verläßlichkeit der Anpassung über Konfidenzintervalle für jeden Parameter sowie die geschätzte Kovarianzmatrix kontrolliert werden. Falls für die numerischen Algorithmen keine guten Startwerte zur Verfügung stehen, können optimierte Startparameter über einem angegebenen Gitter berechnet werden.
Das entwickelte Programmpaket kann auch ohne graphische Oberfläche genutzt werden. Somit ist es möglich, die Algorithmen auf Worksheet-Ebene ebenfalls in früheren Maple-Versionen einzusetzen. In Kapitel 6 werden die erforderlichen Prozeduren mit ihren Parametern beschrieben.

Ein Prototyp des Programmpakets wurde bereits auf dem Maple-Summer-Workshop 2002 in einer Poster-Präsentation [17] an der Universität in Waterloo, Kanada, erfolgreich vorgestellt.

Kapitel 2

Numerische Grundlagen

2.1 Modellierung von Daten

2.1.1 χ^2 Fitting

Grundlage der hier vorgestellten Algorithmen ist ein Datensatz, der aus insgesamt N Punkten besteht. Jeder dieser Punkte besteht dabei aus seinen (ggf. mehrdimensionalen) Koordinaten $\mathbf{x}_i = (x_{i1}, ..., x_{iL}) \in \mathbb{R}^L$ sowie der zugehörigen y-Koordinate y_i und einem ihm zugeordneten Gewicht ω_i:

$$\mathbf{z}_i = (x_{i1}, ..., x_{iL}, y_i, \omega_i) \in \mathbb{R}^{L+2}, \qquad i = 1, ..., N$$

An diese Datenpunkte soll nun eine Modellfunktion y mit M variablen Parametern a_j, $j = 1, ..., M$ angepasst werden. Die hier betrachteten Modellfunktionen haben also folgende Gestalt:

$$\begin{aligned} y : \mathbb{R}^L \times \mathbb{R}^M &\longrightarrow \mathbb{R} \\ \mathbf{x} \longmapsto y(\mathbf{x}) &:= y(x_1, ..., x_L; a_1, ..., a_M) \end{aligned}$$

und sind im Allgemeinen nichtlinear. Die Terme “linear” und “nichtlinear” beziehen sich dabei auf die Abhängigkeit der Modellfunktion von den Parametern $a_1, ..., a_M$. Bei der Modellfunktion $y = a_1 + a_2 \cdot e^{-a_3 \cdot x_1}$ liegt z.B. bei konstantem a_3 ein lineares Fit-Problem vor. Wenn jedoch a_3 ebenfalls variiert wird, entsteht ein nichtlineares Fit-Problem.

Ziel ist es nun, geeignete Werte für die Parameter a_j zu finden. Eine Möglichkeit wäre die Minimierung der Summe der Quadrate der Abweichung zwischen Messwert und Wert der Modellfunktion, also:

$$\min_{a_1,...,a_M} \left(\sum_{i=1}^{N} (y_i - y(\mathbf{x}_i; a_1, ..., a_M))^2 \right) \tag{2.1}$$

Denn man erachtet Parameter $a_1, ..., a_M$ als korrekt, wenn sich mit ihnen die Modellfunktion den Daten sehr gut anschmiegt, d.h. die “Abstände” zwischen vorgegebenem y-Wert und dem Wert $y(\mathbf{x})$ minimal werden.

Dass die Minimierung von (2.1) in der Tat plausible Schätzungen für die Parameter $a_1, ..., a_M$ liefert, kann folgendermaßen begründet werden:

Die Frage, die es zu klären gilt, lautet: “Wie groß ist die Wahrscheinlichkeit, dass bei gegebener Modellfunktion und gegebenen Parametern $a_1, ..., a_M$, die Daten erzeugt werden konnten?”. Da die y-Werte kontinuierlich sind, wird diese Wahrscheinlichkeit immer 0 sein, deshalb der Zusatz “...plus oder minus einem festen Wert $\triangle y$ auf jedem Datenpunkt”. Je kleiner diese Wahrscheinlichkeit ist, desto unwahrscheinlicher sind die Werte der Parameter; je größer sie ist, desto wahrscheinlicher sind die Werte der Parameter.

Diese Wahrscheinlichkeit kann also als Maximum-Likelihood-Schätzwert für die Parameter $a_1, ...,$ a_M betrachtet werden. Unter der Annahme, dass jeder Datenpunkt einen Messfehler hat, der normalverteilt um das “wahre” Modell $y(\mathbf{x})$ ist, und die Standard-Abweichung σ für alle Punkte identisch ist, ergibt sich für diese Wahrscheinlichkeit P:

$$P \sim \prod_{i=1}^{N} \left(\exp(-\frac{1}{2} \cdot \left(\frac{y_i - y(\mathbf{x}_i; a_1, ..., a_M)}{\sigma} \right)^2) \cdot \triangle y \right) \tag{2.2}$$

Nun gelten folgende Beziehungen:

$$\begin{aligned} & \max_{a_1,...,a_M} P \\ \iff & \max_{a_1,...,a_M} log(P) \\ \iff & \min_{a_1,...,a_M} \left(\sum_{i=1}^{N} \frac{(y_i - y(\mathbf{x}_i; a_1, ..., a_M))^2}{2 \cdot \sigma^2} \right) - N \cdot log(\triangle y) \end{aligned}$$

Und dies ist äquivalent zum Minimieren von (2.1), da die Werte N, σ und $\triangle y$ Konstanten sind.

Um nun jedem Punkt $\mathbf{x}_1$, ...,$\mathbf{x}_L$ eine eigene Standardabweichung zuordnen zu können, kann (2.2) durch Ersetzung von σ durch σ_i modifiziert werden. Optimale Parameter erhält man nun durch Minimierung von :

$$\chi^2 : \mathbb{R}^M \longrightarrow \mathbb{R}$$

$$\chi^2(\mathbf{a}) := \sum_{i=1}^{N} \left(\frac{y_i - y(\mathbf{x}_i; a_1, ..., a_M)}{\sigma_i} \right)^2 \tag{2.3}$$

mit $\mathbf{a} = (a_1, ..., a_M)^T \in \mathbb{R}^M$.

2.1.2 Gradient und Hesse-Matrix von χ^2

Die im Folgenden vorgestellten Algorithmen versuchen, wie in 2.1.1 beschrieben, die χ^2 Funktion so weit wie möglich zu minimieren. Dazu werden insbesondere der Gradient und die Hesse-Matrix von χ^2 bezüglich $\mathbf{a}$ benötigt, die hier angegeben werden sollen.
Es gilt:

$$\frac{\partial \chi^2}{\partial a_k}(\mathbf{a}) = -2 \cdot \sum_{i=1}^{N} \left(\frac{y_i - y(\mathbf{x}_i; \mathbf{a})}{\sigma_i^2} \right) \cdot \frac{\partial y(\mathbf{x}_i; \mathbf{a})}{\partial a_k}, \qquad 1 \leq k \leq M$$

Und zudem:

$$\frac{\partial^2 \chi^2}{\partial a_k \partial a_l}(\mathbf{a}) = 2 \sum_{i=1}^{N} \frac{1}{\sigma_i^2} \left(\frac{\partial y(\mathbf{x}_i; \mathbf{a})}{\partial a_k} \cdot \frac{\partial y(\mathbf{x}_i; \mathbf{a})}{\partial a_l} - (y_i - y(\mathbf{x}_i; \mathbf{a})) \cdot \frac{\partial^2 y(\mathbf{x}_i; \mathbf{a})}{\partial a_k \partial a_l} \right) \tag{2.4}$$

$$(1 \leq k, l \leq M)$$

In (2.4) sieht man nun folgendes: Hier treten in der Differenz die zweiten Ableitungen von y auf. Diese werden mit dem Term $(y_i - y(\mathbf{x}_i; \mathbf{a}))$ multipliziert. In der Praxis wird dieser Term vernachlässigt, da die zweiten Ableitungen zum einen in der Berechnung aufwändig sind und zum anderen die Werte dieses Terms vernachlässigbar klein sind gegenüber dem Subtrahend. Obwohl die Vernachlässigung dieses Terms zu einer kleinen Abweichung gegenüber dem exakten Wert führt, soll im weiteren Verlauf das Gleichheitszeichen weiter bestehen bleiben, d.h. (2.4) wird geschrieben als:

$$\frac{\partial^2 \chi^2}{\partial a_k \partial a_l}(\mathbf{a}) = 2 \sum_{i=1}^{N} \frac{1}{\sigma_i^2} \left(\frac{\partial y(\mathbf{x}_i; \mathbf{a})}{\partial a_l} \cdot \frac{\partial y(\mathbf{x}_i; \mathbf{a})}{\partial a_k} \right), \qquad 1 \leq k, l \leq M \tag{2.5}$$

Zusätzlich werden noch definiert:

$$\beta_k := -\frac{1}{2} \cdot \frac{\partial \chi^2}{\partial a_k}(\mathbf{a})$$

und

$$\alpha_{kl} := \frac{1}{2} \cdot \frac{\partial^2 \chi^2}{\partial a_k \partial a_l}(\mathbf{a})$$

Somit ergeben sich:

Gradient von χ^2:

$$\bigtriangledown \chi^2 = -2(\beta_1, ..., \beta_M)^T \tag{2.6}$$

Hesse-Matrix von χ^2:

$$H\chi^2(\mathbf{a}) = 2 \cdot \begin{pmatrix} \alpha_{11} & \dots & \alpha_{1M} \\ \vdots & \ddots & \vdots \\ \alpha_{M1} & \dots & \alpha_{MM} \end{pmatrix} \tag{2.7}$$

2.2 Generelle Algorithmen zur Funktionsminimierung

2.2.1 Methode des steilsten Abstiegs

Eine der grundlegenden Methoden ist die Methode des steilsten Abstiegs.

Sei $f : \mathbb{R}^n \longrightarrow \mathbb{R}$ stetig differenzierbar. Zur Minimierung von f wird nach folgendem Modellalgorithmus vorgegangen:

Steilster Abstieg — Nassi-Shneidermann-Diagramm

Wenn die Bedingung für Konvergenz erfüllt ist, terminiert der Algorithmus.
Berechnung eines Vektors $\mathbf{p}^{(k)}$. (Berechnung einer Suchrichtung)
Berechnung einer Konstanten $\alpha^{(k)}$, so dass $f(\mathbf{x}^{(k)} + \alpha^{(k)} \cdot \mathbf{p}^{(k)}) < f(\mathbf{x}^{(k)})$ erfüllt ist. (Berechnung einer Schrittlänge)
Setze $\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} + \alpha^{(k)} \cdot \mathbf{p}^{(k)}$. (Bestimmung des nächsten Iterationspunktes)

Wichtige Faktoren hierbei sind die Bestimmung der Suchrichtung und der Schrittlänge.

Um die Suchrichtung zu bestimmen wird folgendermaßen vorgegangen: Bricht man die Taylorentwicklung von f um $\mathbf{x}^{(k)}$ nach dem zweiten Glied ab, so ergibt sich:

$$f(\mathbf{x}^{(k)} + \mathbf{p}) \approx f(\mathbf{x}^{(k)}) + \bigtriangledown f(\mathbf{x}^{(k)})^T \cdot \mathbf{p}$$

Hieran ist zu sehen, dass bei einer Minimierung von f der zweite Summand möglichst so zu wählen ist, dass sich eine große negative Zahl ergibt. Offensichtlich muss das gewählte $\mathbf{p}$ vorher normiert werden, denn falls $\bigtriangledown f(\mathbf{x}^{(k)})^T \cdot \bar{\mathbf{p}} < 0$, könnte $\mathbf{p}$ einfach als $a \cdot \bar{\mathbf{p}}$ gewählt werden, wobei a eine positive Konstante ist.
Mit einer gegebenen Norm $||.||$ geht es also um folgende Minimierung:

$$\min_{\mathbf{p} \in \mathbb{R}^n} \frac{\bigtriangledown f(\mathbf{x}^{(k)})^T \cdot \mathbf{p}}{||\mathbf{p}||}.$$

Bei Verwendung der euklidischen Norm $||\mathbf{p}|| = \sqrt{(\mathbf{p}^T \cdot \mathbf{p})}$ ergibt sich als Suchrichtung gerade der negative Gradient, also

$$\mathbf{p}^{(k)} = - \bigtriangledown f(\mathbf{x}^{(k)})$$

Bei der Bestimmung der Schrittlänge kann kein allgemein gültiges Verfahren angegeben werden, wodurch die Methode des steilsten Abstiegs bis hierhin zunächst die Gestalt

$$\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} - c \cdot \bigtriangledown f(\mathbf{x}^{(k)}) \tag{2.8}$$

mit $c \in \mathbb{R}$ besitzt.
Methoden dieser Art haben den Vorteil, dass sie sehr leicht zu implementieren sind. Andererseits sind sie wegen ihres langsamen Konvergenzverhaltens zur Lösung praktischer Probleme alleine nicht relevant. [2] gibt hierzu noch weitere Informationen.

2.2.2 Verfahren von Newton

Sei $f : \mathbb{R}^n \longrightarrow \mathbb{R}$ als zweimal stetig differenzierbare Funktion gegeben, deren Minimum gesucht ist, und $\mathbf{x}^{(0)} \in \mathbb{R}^n$ der gegebene Startwert.

Unter den obigen Voraussetzungen wird eine Folge von Punkten $\{\mathbf{x}^{(k)}\}$ mit Startpunkt $\mathbf{x}^{(0)}$ nach folgendem Schema generiert:

$$(Hf(\mathbf{x}^{(k)}))(\mathbf{x}^{(k+1)} - \mathbf{x}^{(k)}) = - \bigtriangledown f(\mathbf{x}^{(k)}), \qquad k \geq 0 \tag{2.9}$$

oder, falls H regulär ist, durch

$$\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} - (Hf(\mathbf{x}^{(k)})^{-1} \bigtriangledown f(\mathbf{x}^{(k)}), \qquad k \geq 0 \tag{2.10}$$

Die Herleitung kann in der folgenden Weise angegeben werden: Nach der Taylor'schen Formel kann f lokal durch die Funktion f_k mit

$$f_k(\mathbf{x}) = f(\mathbf{x}^{(k)}) + \bigtriangledown f(\mathbf{x}^{(k)})^T \cdot (\mathbf{x} - \mathbf{x}^{(k)}) + \frac{1}{2} \cdot (\mathbf{x} - \mathbf{x}^{(k)})^T Hf(\mathbf{x}^{(k)}) \cdot (\mathbf{x} - \mathbf{x}^{(k)})$$

approximiert werden.

Wenn $\mathbf{x}^{(k)}$ der k-te Term der Punktefolge ist, welche gegen einen Punkt $\mathbf{x}^*$ konvergieren soll, wird als nächster Punkt $\mathbf{x}^{(k+1)}$ der Minimierer von f_k genommen, falls dieser existiert. Dass so ein Punkt existiert, falls die Hesse-Matrix $Hf(\mathbf{x}^{(k)})$ positiv definit ist, zeigt der Satz 2.2.1.
Wenn also $\mathbf{y}^*$ dieser Punkt von f_k ist, dann gilt:

$$0 = \bigtriangledown f_k(\mathbf{y}^*) = \bigtriangledown f(\mathbf{x}^{(k)}) + Hf(\mathbf{x}^{(k)}) \cdot (\mathbf{y}^* - \mathbf{x}^{(k)})$$

und das ist gerade äquivalent zu (2.9), wenn $\mathbf{y}^*$ durch $\mathbf{x}^{(k+1)}$ ersetzt wird.

Satz 2.2.1 Sei A eine positiv definite $n \times n$ Matrix; $\mathbf{b}, \mathbf{x} \in \mathbb{R}^n, a \in \mathbb{R}$
Dann ist die quadratische Funktion

$$f(\mathbf{x}) = a + \mathbf{b}^T \cdot \mathbf{x} + \frac{1}{2} \cdot \mathbf{x}^T \cdot A \cdot \mathbf{x}$$

strikt konvex und besitzt einen eindeutigen globalen Minimierer im Punkt $\mathbf{x}^*$, der die eindeutige Lösung des linearen Gleichungssystems

$$A \cdot \mathbf{x} = -\mathbf{b}$$

ist. Die Newtonsche Punktefolge mit Startpunkt $\mathbf{x}^{(0)}$ erreicht diesen Punkt in einem Schritt, d.h.

$$\mathbf{x}^{(1)} = \mathbf{x}^*$$

Beweis:
Zunächst gilt $\bigtriangledown f(\mathbf{x}) = \mathbf{b} + A \cdot \mathbf{x}$ und $Hf(\mathbf{x}) = A$.

Da A positiv definit ist, folgt daraus, dass f strikt konvex ist, und einen globalen Minimierer $\mathbf{x}^*$ besitzt, der Lösung des linearen Gleichungssystems $A \cdot \mathbf{x} = -\mathbf{b}$ ist (für einen Beweis siehe z.B. [3]).

Zu zeigen ist noch, dass das Newtonsche Iterationsverfahren diesen Punkt in einem Schritt erreicht. Es gilt:

$$\begin{aligned}
\mathbf{x}^{(1)} &= \mathbf{x}^{(0)} - (Hf(\mathbf{x}^{(0)}))^{-1} \cdot \bigtriangledown f(\mathbf{x}^{(0)}) \\
&= \mathbf{x}^{(0)} - A^{-1}(\mathbf{b} + A \cdot \mathbf{x}^{(0)}) \\
&= \mathbf{x}^{(0)} - A^{-1}(-A \cdot \mathbf{x}^* + A \cdot \mathbf{x}^{(0)}) \\
&= \mathbf{x}^{(0)} + \mathbf{x}^* - \mathbf{x}^{(0)} \\
&= \mathbf{x}^*
\end{aligned}$$

□

Das Newton-Verfahren eignet sich besonders dann, wenn der Startpunkt $\mathbf{x}^{(0)}$ nahe am Minimum $\mathbf{x}^*$ liegt und $H(\mathbf{x}^*)$ regulär ist. Unter dieser Voraussetzung konvergiert das Verfahren quadratisch.

2.2.3 Restriktive Schrittverfahren

Das Newton-Verfahren (2.10) liefert sehr gute Ergebnisse, falls die Hesse-Matrix $Hf(\mathbf{x}^{(k)})$ positiv definit ist. Ist dies nicht der Fall, hat die betrachtete Taylor-Funktion $f_k(\mathbf{x})$ kein eindeutiges globales Minimum und die Methode ist nicht mehr definiert. Eine Möglichkeit, den nächsten Iterationspunkt zu finden, ist nun f in einer Umgebung $\Omega^{(k)}$ von $\mathbf{x}^{(k)}$ zu betrachten. In dieser Umgebung wird derjenige Punkt $\mathbf{y}^{(k)}$ gesucht, der $f_k(\mathbf{x})$ minimiert. Als nächster Iterationspunkt wird nun

$$\mathbf{x}^{(k+1)} = \mathbf{y}^{(k)} \tag{2.11}$$

gesetzt.

Algorithmen, die nach einem solchen Schema arbeiten, nennt man allgemein restriktive Schrittverfahren. Sie haben den Vorteil, dass die schnelle Konvergenzgeschwindigkeit des Newton-Verfahrens beibehalten wird. Es stellt sich die Frage, wie die Umgebung $\Omega^{(k)}$ zu wählen ist. Hier macht es Sinn, den Fall

$$\Omega^{(k)} = \{\mathbf{x} : ||\mathbf{x} - \mathbf{x}^{(\mathbf{k})}|| \leq h^{(k)}\}$$

zu betrachten. Es wird diejenige Lösung $\mathbf{y}^{(k)}$ gesucht, die die Taylor-Funktion in der Umgebung $\Omega^{(k)}$ minimiert. Also wird die Lösung des Unterproblems

$$\mathbf{y}^{(k)} = \min_{\mathbf{z} \in \Omega^{(k)}} f_k(\mathbf{z}) \tag{2.12}$$

gesucht und der neue Iterationspunkt dann wie in (2.11) gesetzt.

Die Minimierung von (2.12) ist abhängig von der gewählten Norm und der Wahl des Radius $h^{(k)}$. Um eine übertriebene Einschränkung zu vermeiden, sollte $h^{(k)}$ zunächst so groß wie möglich gewählt werden. Um ein Maß für die Wahl zu erhalten, wird zunächst

$$\triangle f^{(k)} = f(\mathbf{x}^{(k)}) - f(\mathbf{y}^{(k)}) \tag{2.13}$$

und

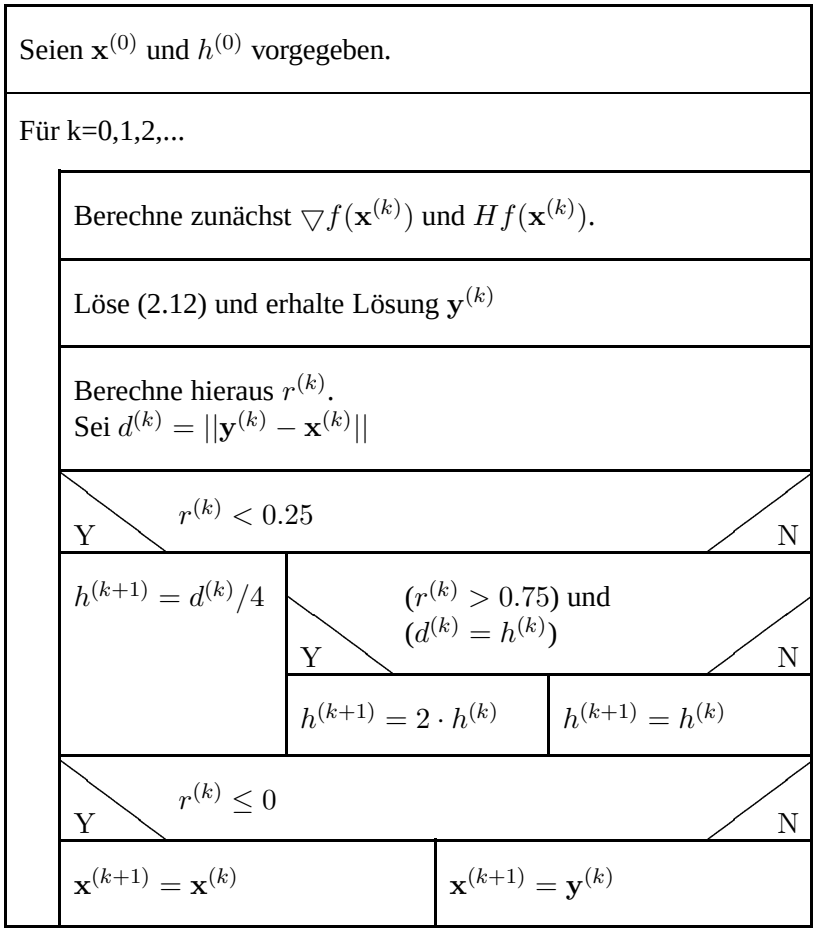
$$\triangle q^{(k)} = f_k(\mathbf{x}^{(k)}) - f_k(\mathbf{y}^{(k)}) = f(\mathbf{x}^{(k)}) - f_k(\mathbf{y}^{(k)}) \tag{2.14}$$

gesetzt. Betrachtet man nun den Quotienten von (2.13) und (2.14), also

$$r^{(k)} = \frac{\triangle f^{(k)}}{\triangle q^{(k)}},$$

dann misst dieser Quotient die Approximation von f durch f_k. Die Approximation ist umso besser, je näher dieser Quotient an 1 liegt, und umso schlechter, je mehr er von 1 abweicht. Mit diesen Informationen kann nun ein prototypischer Algorithmus geschrieben werden.

Restriktives Schrittverfahren — Nassi-Shneidermann-Diagramm



Im 4. Schritt wird der Radius h aufgrund der Übereinstimmung von f und f_k gewählt. War die Approximation gut und lag der Punkt $\mathbf{y}^{(k)}$ auf dem Kreisradius, dann wird der bisherige Radius verdoppelt.
War die Approximation schlecht, wird als neuer Radius der Abstand von $x^{(k)}$ und $y^{(k)}$ dividiert durch 4 genommen. In allen anderen Fällen wird der Radius beibehalten.
Die Konstanten in diesem Algorithmus sind hier willkürlich gewählt und werden bei Modifikation das Verhalten sehr stark beeinflussen.
Wie in [1] gezeigt wird, sind solche Verfahren unter bestimmten Voraussetzungen global konvergent und besitzen ebenfalls die Konvergenzordnung 2.
Der prototypische Algorithmus kann in der gegebenen Form nicht implementiert werden. Für die Implementierung muß noch eine Norm und ein Verfahren zu Minimierung von (2.12) spezifiziert werden. Er ist Grundlage für den Levenberg-Marquardt-Algorithmus in (2.3.3).

2.3 Spezielle Algorithmen zur Funktionsminimierung

Wie im Abschnitt 2.1.2 gesehen, haben die Funktionen, die bei der Anpassung von Daten vorkommen, alle eine gemeinsame Gestalt. Die Funktion χ^2 besteht aus einer Summe von N nicht zwingend linearen Funktionen r_i^2, d.h.

$$\chi^2(\mathbf{x}) = \sum_{i=1}^{N} r_i(\mathbf{x})^2.$$

Diese Funktionen können einerseits durch die in Kapitel 2.2 beschriebenen Methoden minimiert werden. Im Nachfolgenden sollen jedoch spezielle Algorithmen für diesen Typ von Funktionen vorgestellt werden. Zuvor werden jedoch noch Eigenschaften dieses Funktionstyps aufgezeigt.

Allgemein wird hier eine Funktion

$$f : \mathbb{R}^n \longrightarrow \mathbb{R}$$
$$f(\mathbf{x}) = \sum_{i=1}^{m} (r_i(\mathbf{x}))^2 = ||\mathbf{r}||_2^2$$

mit

$$r_i : \mathbb{R}^n \longrightarrow \mathbb{R}$$
$$\mathbf{r}(\mathbf{x}) = (r_1(\mathbf{x}), ..., r_m(\mathbf{x}))^T \quad , 1 \leq i \leq m$$

betrachtet, die es zu minimieren gilt.

Für den Gradienten dieser Funktion gilt:

$$\bigtriangledown f = 2 \cdot \mathbf{A}^T \cdot \mathbf{r} \tag{2.15}$$

wobei $\mathbf{A}$ die Jacobi-Matrix von $\mathbf{r}$ darstellt, mit

$$A = \begin{pmatrix} \frac{\partial r_1}{\partial x_1} & \cdots & \frac{\partial r_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial r_m}{\partial x_1} & \cdots & \frac{\partial r_m}{\partial x_n} \end{pmatrix}$$

Für die Hesse-Matrix von f ergibt sich:

$$Hf = 2 \cdot \mathbf{A}^T\mathbf{A} + 2\sum_{i=1}^{m} r_i \cdot Hr_i \tag{2.16}$$

wobei Hr_i die Hesse-Matrix der Funktion r_i bezeichnet.

2.3.1 Der Gauß-Newton-Algorithmus

Der Gauß-Newton-Algorithmus basiert auf der speziellen Struktur der zu minimierenden Funktion f aus Abschnitt 2.3. Aus (2.10) ist ersichtlich, dass zur Berechnung des nächsten Iterationspunktes $\mathbf{x}^{(k+1)}$ die inverse Hesse-Matrix von f benötigt wird. Bei der praktischen Realisierung des Newton-Verfahrens wird man sich deshalb auf die äquivalente Form (2.9) beziehen. Definiert man den Vektor

$$\mathbf{p}^{(k)} = \mathbf{x}^{(k+1)} - \mathbf{x}^{(k)},$$

dann kann das Newton-Verfahren ebenfalls in der Form

$$Hf(\mathbf{x}^{(k)}) \cdot \mathbf{p}^{(k)} = - \bigtriangledown f(\mathbf{x}^{(k)}) \tag{2.17}$$

geschrieben werden. Den Vektor $\mathbf{p}^{(k)}$ bezeichnet man auch als die Newton-Richtung. Zur Lösung von (2.17) ist ein lineares Gleichungssystem zu lösen.

Benutzt man die spezielle Form des Gradienten in (2.15) und der Hesse-Matrix in (2.16), so erhält man aus (2.17) die Beziehung

$$\left(\mathbf{A}^T(\mathbf{x}^{(k)})\mathbf{A}(\mathbf{x}^{(k)}) + \sum_{i=1}^{m} r_i(\mathbf{x}^{(k)})Hr_i(\mathbf{x}^{(k)})\right)\mathbf{p}^{(k)} = -\mathbf{A}^T(\mathbf{x}^{(k)})\mathbf{r}(\mathbf{x}^{(k)}) \tag{2.18}$$

Unter der Annahme, dass $||\mathbf{r}(\mathbf{x}^{(k)})|| \to 0$, wenn sich $\mathbf{x}^{(k)}$ der Lösung nähert, geht damit auch die Matrix $\sum_{i=1}^{m} r_i(\mathbf{x}^{(k)})Hr_i(\mathbf{x}^{(k)})$ gegen Null. Damit kann (2.18) approximiert werden durch eine Lösung des Systems:

$$(\mathbf{A}^T(\mathbf{x}^{(k)})\mathbf{A}(\mathbf{x}^{(k)}))\mathbf{p}^{(k)} = -\mathbf{A}^T(\mathbf{x}^{(k)})\mathbf{r}(\mathbf{x}^{(k)}) \tag{2.19}$$

also durch $-\mathbf{A}_k^+\mathbf{r}(\mathbf{x}^{(k)})$, wobei $\mathbf{A}_k^+$ die Pseudoinverse zu $\mathbf{A}(\mathbf{x}^{(k)})$ ist. Hat $\mathbf{A}(\mathbf{x}^{(k)})$ vollen Rang, so besitzt (2.19) eine eindeutige Lösung und $\mathbf{A}_k^+$ ist gegeben durch

$$\mathbf{A}_k^+ = (\mathbf{A}^T(\mathbf{x}^{(k)})\mathbf{A}(\mathbf{x}^{(k)}))^{-1}\mathbf{A}^T(\mathbf{x}^{(k)}).$$

Nach der Definition von $\mathbf{p}^{(k)}$ kann der nächste Iterationspunkt $\mathbf{x}^{(k+1)}$ geschrieben werden als:

$$\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} - \mathbf{A}_k^+\mathbf{r}(\mathbf{x}^{(k)})$$

Die Lösung $\mathbf{p}^{(k)}$ von (2.19) ist eine Lösung des linearen "least-squares" Problems

$$\min_{\mathbf{p}\in\mathbb{R}^n} ||\mathbf{A}(\mathbf{x}^{(k)})\mathbf{p} + \mathbf{r}(\mathbf{x}^{(\mathbf{k})})||_2^2 \tag{2.20}$$

und der Vektor $\mathbf{p}$, der durch Lösen von (2.20) bestimmt wird, heißt die Gauß-Newton-Richtung.

Der wesentliche Vorteil ist dabei, dass zur Berechnung des Gradienten nur $\mathbf{A}$ und $\mathbf{r}$ benutzt werden. Quasi-Newton-Methoden (die hier nicht weiter beschrieben werden) benötigen für diese Approximation ggf. bis zu n Iterationen. Hier ist die Approximation sofort verfügbar, wodurch eventuell eine Beschleunigung der Konvergenz zu erwarten ist.

Aufgrund der Approximation der Hesse-Matrix ist das Gauss-Newton-Verfahren allerdings nur dann sinnvoll, wenn der Startwert $\mathbf{x}^{(0)}$ nahe beim Minimum $\mathbf{x}^*$ liegt und die Werte r_i klein sind oder der Grad der Nichtlinearität bei der Approximation klein ist. Weitere Informationen zum Gauss-Newton-Algorithmus finden sich insbesondere in [2] (S. 134-136).

2.3.2 Modifizierter Gauß-Newton-Algorithmus

Der hier vorgestellte modifizierte Gauß-Newton-Algorithmus aus [18] ist eine Kombination des Gauß-Newton-Verfahrens mit einem Quasi-Newton-Verfahren. Betrachtet wird hier das Iterationsschema:

$$\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} + \alpha^{(k)}\mathbf{p}^{(k)}$$

Hier bezeichnet $\mathbf{p}^{(k)}$ die Suchrichtung und $\alpha^{(k)}$ ist eine Konstante, die so gewählt wird, dass $f(\mathbf{x}^{(k)} + \alpha^{(k)}\mathbf{p}^{(k)})$ minimal wird.
Die Wahl der Suchrichtung wird in diesem Algorithmus nun variiert: Hat die letzte Iteration eine deutliche Reduktion des Funktionswertes gebracht, dann wird als Suchrichtung (2.20) genommen. Andernfalls wird nach einem Quasi-Newton-Verfahren gearbeitet, das in [2] ausführlich beschrieben wird.

2.3.3 Levenberg-Marquardt-Methode

Im Folgenden soll ein weiteres Verfahren zum Lösen nichtlinearer Ausgleichsprobleme beschrieben werden, das Levenberg-Marquardt-Verfahren. Dieses beruht auf dem Gauß-Newton-Verfahren aus Kapitel 2.3.1 und dem im Kapitel 2.2.3 beschriebenen restriktiven Schrittverfahren. Das Gauß-Newton-Verfahren (2.20) wird durch die zusätzliche Forderung $||\mathbf{p}||_2 \leq h^{(k)}$ modifiziert, d.h. es ist die Lösung von

$$\min_{\mathbf{p}} ||\mathbf{A}(\mathbf{x}^{(k)})\mathbf{p} + \mathbf{r}(\mathbf{x}^{(k)})||_2^2$$

unter der Bedingung

$$||\mathbf{p}||_2 \leq h^{(k)}$$

gesucht. Das Bestimmen der nächsten Iterierten $\mathbf{x}^{(k+1)}$ wird also auf eine Kugel-Umgebung von $\mathbf{x}^{(k)}$ beschränkt (der sogenannten trust region).

Mit $\mathbf{b}^{(k)} = -\mathbf{r}(\mathbf{x}^{(k)})$ und $\Psi(\mathbf{p}) = \frac{1}{2}||\mathbf{A}(\mathbf{x}^{(k)})\mathbf{p} - \mathbf{b}^{(k)}||_2^2$ ergibt sich das restringierte Minimierungsproblem als

$$\min_{\mathbf{p}} \Psi(\mathbf{p}) \tag{2.21}$$

unter der Nebenbedingung

$$||\mathbf{p}||_2 \leq h^{(k)}$$

Ist eine Lösung $\mathbf{p}^{(k)}$ des Problems gefunden, ergibt sich der nächste Iterationspunkt als

$$\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} + \mathbf{p}^{(k)}$$

Nun können die folgenden beiden Fälle unterschieden werden:

(a) $||\mathbf{p}^{(k)}||_2 < h^{(k)}$
$\mathbf{p}^{(k)}$ ist dann ein stationärer Punkt von $\Psi(\mathbf{p})$, erfüllt also die Bedingung $\bigtriangledown\Psi(\mathbf{p}) = 0$. Um den Gradienten von Ψ zu berechnen, betrachtet man zunächst die Ableitung von Ψ in Richtung $\mathbf{d}$:

$$\frac{\partial}{\partial \mathbf{d}}\Psi = \lim_{t\to 0} \frac{\Psi(\mathbf{x} + t\mathbf{d}) - \Psi(\mathbf{x})}{t}$$

Es gilt ($\mathbf{A}(\mathbf{x}^{(k)})$ wird durch $\mathbf{A}_k$ abgekürzt):

$$\Psi(\mathbf{x} + t\mathbf{d}) - \Psi(\mathbf{x})$$

$$= \frac{1}{2}||\mathbf{b}^{(k)} - \mathbf{A}_k(\mathbf{x} + t\mathbf{d})||_2^2 - \frac{1}{2}||\mathbf{b}^{(k)} - \mathbf{A}_k\mathbf{x}||_2^2$$

$$= \frac{1}{2}(\mathbf{b}^{(k)} - \mathbf{A}_k\mathbf{x} - t\mathbf{A}_k\mathbf{d})^T(\mathbf{b}^{(k)} - \mathbf{A}_k\mathbf{x} - t\mathbf{A}_k\mathbf{d}) - \frac{1}{2}(\mathbf{b}^{(k)} - \mathbf{A}_k\mathbf{x})^T(\mathbf{b}^{(k)} - \mathbf{A}_k\mathbf{x})$$

Nach dem Ausmultiplizieren und Zusammenfassen ergibt sich:

$$t(\mathbf{A}_k\mathbf{d})^T(\mathbf{A}_k\mathbf{x} - \mathbf{b}^{(k)}) + \frac{1}{2}t^2(\mathbf{A}_k\mathbf{d})^T(\mathbf{A}_k\mathbf{d})$$

Für die Richtungsableitung folgt somit:

$$\frac{\partial}{\partial\mathbf{d}}\Psi(\mathbf{x}) = (\mathbf{A}_k\mathbf{d})^T(\mathbf{A}_k\mathbf{x} - \mathbf{b}^{(k)})$$

Da $(\mathbf{A}_k\mathbf{d})^T(\mathbf{A}_k\mathbf{x} - \mathbf{b}^{(k)}) = ((\mathbf{A}_k\mathbf{d})^T(\mathbf{A}_k\mathbf{x} - \mathbf{b}^{(k)}))^T = (\mathbf{A}_k\mathbf{x} - \mathbf{b}^{(k)})^T(\mathbf{A}_k\mathbf{d})$ für alle $\mathbf{d} \in \mathbb{R}^n$ ist, ergibt sich für den Gradienten von Ψ:

$$\bigtriangledown\Psi(\mathbf{x}) = \mathbf{A}_k^T(\mathbf{A}_k\mathbf{x} - \mathbf{b}^{(k)})$$

Die Lösung $\mathbf{p}^{(k)}$ erfüllt also die Gaußschen Normalgleichungen

$$\mathbf{A}_k^T\mathbf{A}_k\mathbf{x} = \mathbf{A}_k^T\mathbf{b}^{(k)}.$$

(b) $||\mathbf{p}^{(k)}||_2 = h^{(k)}$
In diesem Fall muss der Gradient von Ψ an der Stelle $\mathbf{p}^{(k)}$ auf den Mittelpunkt des Kreises $\{\mathbf{p} \mid ||\mathbf{p}||_2 \leq h^{(k)}\}$ zeigen, wie in [6] weiter erläutert wird. Es existiert also eine Darstellung der Form

$$\bigtriangledown\Psi(\mathbf{p}^{(k)}) = \mathbf{A}_k^T\mathbf{A}_k\mathbf{p}^{(k)} - \mathbf{A}_k^T\mathbf{b}^{(k)} = -\lambda_k\mathbf{p}^{(k)}$$

für ein $\lambda_k > 0$. Das ist äquivalent zu

$$(\mathbf{A}_k^T\mathbf{A}_k + \lambda_k\mathbf{I})\mathbf{p}^{(k)} = \mathbf{A}_k^T\mathbf{b}^{(k)}$$

wobei $\mathbf{I}$ die $n \times n$ Einheitsmatrix bezeichnet.

In beiden Fällen genügt die Lösung $\mathbf{p} = \mathbf{p}^{(k)}$ der Gleichung

$$(\mathbf{A}_k^T\mathbf{A}_k + \lambda_k I)\mathbf{p} = \mathbf{A}_k^T\mathbf{b}^{(k)} \tag{2.22}$$

für ein $\lambda_k \geq 0$. Dieses λ_k ist genau dann positiv, wenn $||\mathbf{p}^{(k)}|| = h^{(k)} > 0$ gilt.

Der Parameter $\mathbf{p}^{(k)}$ muss nun noch numerisch bestimmt werden. Dazu fasst man (2.22) als eine einparametrige Schar linearer Gleichungen mit Parameter λ und Lösung $\mathbf{p} = \mathbf{p}_\lambda$ auf. Dann wird der Parameter $\lambda = \lambda_k$ und die zugehörige Lösung $\mathbf{p}^{(k)} = \mathbf{p}_\lambda$ gesucht, für die die zugehörige Nebenbedingung

$$||\mathbf{p}||_2 = ||(\mathbf{A}_k^T\mathbf{A}_k + \lambda\mathbf{I})^{-1}\mathbf{A}_k^T\mathbf{b}^{(k)}||_2 = h^{(k)} \tag{2.23}$$

erfüllt ist, falls so ein positiver Parameter λ_k existiert.
Die Matrix $\mathbf{A}_k^T\mathbf{A}_k$ ist symmetrisch und positiv semidefinit. Seien $\mathbf{u}_i$ die orthonormierten Eigenvektoren, und d_i die zugehörigen, nicht negativen Eigenwerte. Der Vektor $\mathbf{A}_k^T\mathbf{b}^{(k)}$ kann dann als Linearkombination geschrieben werden:

$$\mathbf{A}_k^T\mathbf{b}^{(k)} = \sum_{i=1}^{n} z_i\mathbf{u}_i$$

mit Konstanten $z_i \in \mathbb{R}$. Damit ergibt sich:

$$\mathbf{p}_\lambda = (\mathbf{A}_k^T \mathbf{A}_k + \lambda \mathbf{I})^{-1} \mathbf{A}_k^T \mathbf{b}^{(k)} = \sum_{i=1}^{n} \frac{z_i}{d_i + \lambda} \mathbf{u}_i$$

Aus (2.23) folgt dann:

$$r(\lambda) = \sum_{i=1}^{n} \frac{z_i^2}{(d_i + \lambda)^2} = (h^{(k)})^2$$

Der Parameter λ kann nun mittels des Hebden-Verfahrens berechnet werden. Das Hebden-Verfahren ist ein spezielles Verfahren zum Lösen von nichtlinearen Gleichungen des obigen Typs und wird in [6] eingehend beschrieben.

Als nächstes ist ein Verfahren zur Wahl des trust region Radius $h^{(k)}$ gesucht. Wie bereits in 2.2.3 beschrieben wurde, sollte die Wahl so durchgeführt werden, dass dieser Radius zunächst so groß wie möglich gewählt wird. Dabei sollte dieser Wert allerdings nicht zu groß werden, um den Fehler in der Linearisierung in einem angemessenen Rahmen zu halten. In Kapitel (2.2.3) wurde bereits ein Kriterium zur Abschätzung der Linearität angegeben, das jetzt verschärft wird (Armijo-Goldstein-Kriterium). Eine neue Näherung $\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} + \mathbf{p}^{(k)}$ wird genau dann akzeptiert, wenn

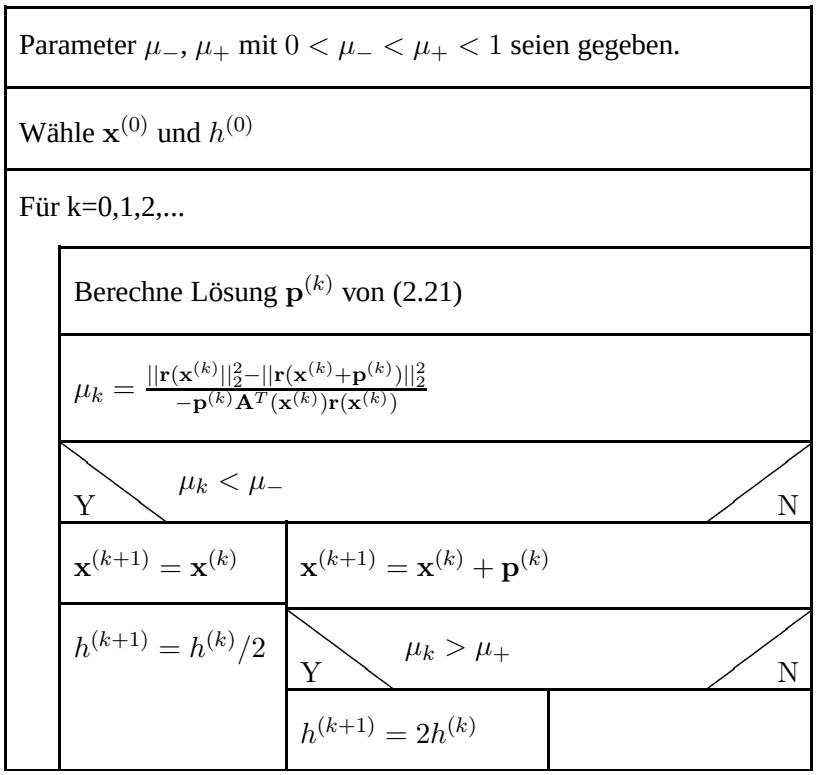
$$\frac{1}{2} \frac{f(\mathbf{x}^{(k)}) - f(\mathbf{x}^{(k)} + \mathbf{p}^{(k)})}{| \bigtriangledown f(\mathbf{x}^{(k)})^T \mathbf{p}^{(k)} |} \geq \mu$$

für einen Parameter $\mu \in (0, 1)$ erfüllt ist ([6]). Je größer μ gewählt werden kann, desto erfolgreicher ist der Iterationsschritt. Mit (2.15) kann dieser Bruch in die Darstellung

$$\mu_k = \frac{||\mathbf{r}(\mathbf{x}^{(k)}||_2^2 - ||\mathbf{r}(\mathbf{x}^{(k)} + \mathbf{p}^{(k)})||_2^2}{-\mathbf{p}^{(k)} \mathbf{A}^T(\mathbf{x}^{(k)}) \mathbf{r}(\mathbf{x}^{(k)})}$$

gebracht werden. Für die Anwendung des Armijo-Goldstein-Kriteriums werden zwei Toleranzschranken μ_- und μ_+ benötigt mit $0 < \mu_- < \mu_+ < 1$. Dabei wird die neue Iterierte $\mathbf{x}^{(k+1)}$ akzeptiert, falls $\mu_k \geq \mu_-$ ist. Der Iterationsschritt war in diesem Fall erfolgreich, und der Radius der trust region wird für den nächsten Iterationsschritt beibehalten. Falls sogar $\mu_k > \mu_+$ gilt, könnte dieser Radius sogar verdoppelt werden, da sich die Funktion um die Iterierte fast linear verhält. Gilt jedoch $\mu_k < \mu_-$, dann wird die berechnete Iterierte nicht als neue Näherung akzeptiert. In diesem Fall war der Radius zu groß gewählt, und der Iterationsschritt wird mit dem halbierten Radius wiederholt.

Levenberg-Marquardt-Algorithmus — Nassi-Shneidermann-Diagramm



Benötigt wird nun noch ein Kriterium zum Abbruch des Iterationsprozesses. Eine Möglichkeit besteht darin, den Iterationsprozeß zu stoppen, falls sich der relative Wert von f nach drei Iterationen nicht mehr ändert, also $\frac{|f(\mathbf{x}^{(k+1)}) - f(\mathbf{x}^{(k)})|}{f(\mathbf{x}^{(k)})} < 0.00001, k = n, n+1, n+2$

2.3.4 Beispiele

Als Beispiel für das Auffinden von Parametern wird hier die nichtlineare Modellfunktion

$$y : \mathbb{R} \longrightarrow \mathbb{R}$$

$$y(x) = a_1 \cdot (1 - \exp(-a_2 \cdot x))$$

betrachtet, wobei die Daten aus Tabelle 2.1 zugrunde liegen.

Nr.	x	y	ω
1	77.6	10.07	1
2	114.9	14.73	1
3	141.1	17.94	1
4	190.8	23.93	1
5	239.9	29.61	1
6	289.0	35.18	1
7	332.8	40.02	1
8	378.4	44.82	1
9	434.8	50.76	1
10	477.3	55.05	1
11	536.8	61.01	1
12	593.1	66.40	1
13	689.1	75.47	1
14	760.0	81.78	1

Tabelle 2.1: Datensatz Misra1a [7]

Die Startwerte, die man ggf. nur durch Raten erhält, sollen zunächst $a_1 = 500$ und $a_2 = 0.0008$ betragen. Als Algorithmus zur Bestimmung der Parameter soll hier der Levenberg-Marquardt-Algorithmus verwendet werden.

Die Funktion χ^2 hat die Gestalt:

$$\chi^2 : \mathbb{R}^2 \longrightarrow \mathbb{R}$$

$$\chi^2(a_1, a_2) = \sum_{i=1}^{14} (y_i - a_1(1 - \exp(-a_2 \cdot x_i)))^2 \tag{2.24}$$

Da die Funktion χ^2 in diesem einfachen Beispiel nur von 2 Parametern abhängt, kann sie leicht graphisch veranschaulicht werden. Abbildung 2.1 zeigt den Verlauf dieser Funktion ausschnittsweise.

Mit dem Levenberg-Marquardt-Algorithmus wird nun ein Minimum dieser Funktion gesucht. Als Iterationspunkte ergeben sich die in Tabelle 2.2 angegebenen Werte. Wie man sieht, wird das Minimum nach nur vier Iterationen erreicht.

Nr.	a_1	a_2	$\chi^2(a_1, a_2)$
1	500	0.0008	113736.562
2	250.974	0.000649	1429.852
3	234.036	0.000568	1.606
4	238.698	0.000550	0.177
5	238.942	0.000550	0.126
6	238.942	0.000550	0.126

Tabelle 2.2: Näherungswerte für Startwerte $a_1 = 500$, $a_2 = 0.0008$

Abbildung 2.2 zeigt die Iterationspunkte graphisch.

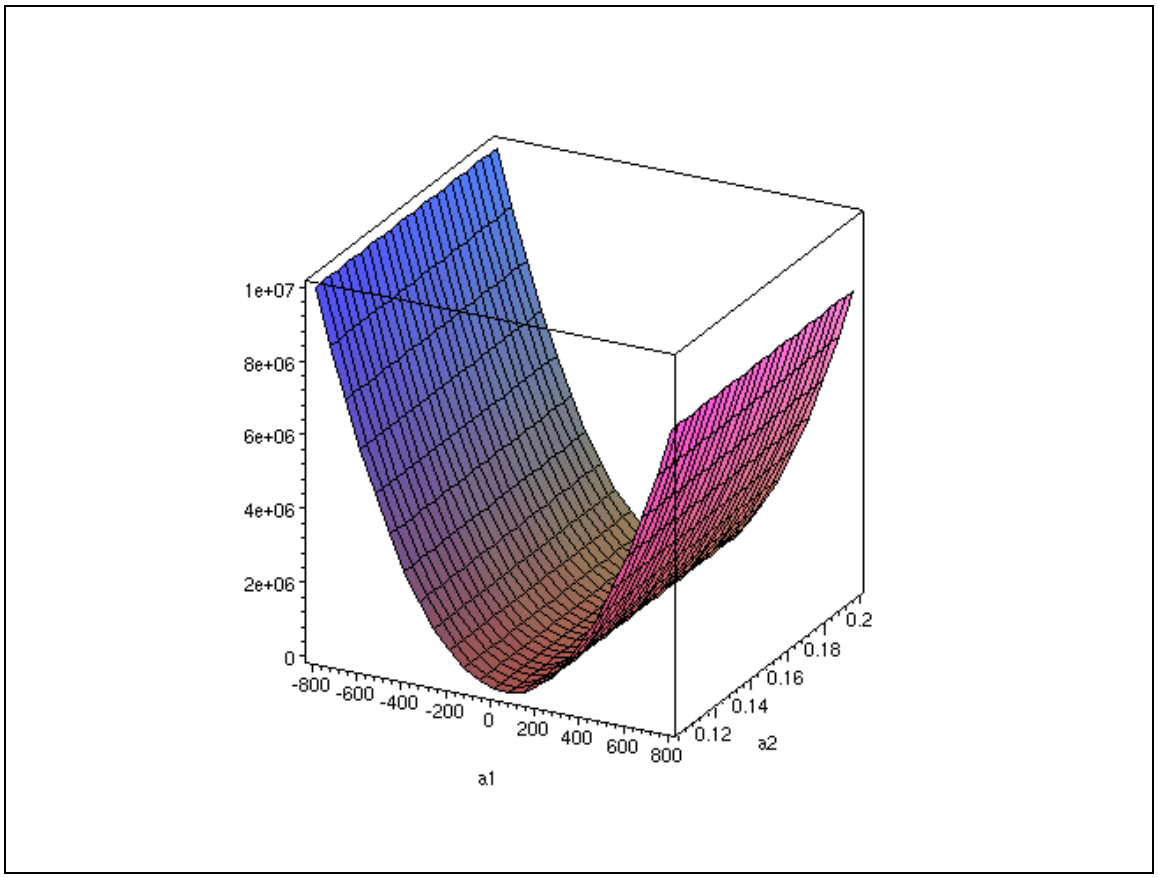


Abbildung 2.1: Die Beispielfunktion (2.24)

Die Wahl anderer Startwerte kann zu einem unterschiedlichen Verhalten des Algorithmus führen. Als zweites Beispiel seien hier die Startwerte $a_1 = 500$ und $a_2 = 0.0001$ gewählt. Ein Ausschnitt aus den Iterationspunkten ist in Tabelle 2.3 wiedergegeben.

Nr.	a_1	a_2	$\chi^2(a_1, a_2)$
1	500	0.0001	10780.190
2	674.168	0.000201	602.543
3	507.606	0.000229	64.667
4	504.860	0.000240	19.873
5	492.574	0.000246	19.027
6	480.550	0.000253	18.211
...	...	...	...
33	272.941	0.000472	1.338
34	254.525	0.000508	1.476
35	239.195	0.000547	0.621
36	238.942	0.000550	0.126
37	238.942	0.000550	0.126

Tabelle 2.3: Iterationspunkte für Startwerte $a_1 = 500$, $a_2 = 0.0001$

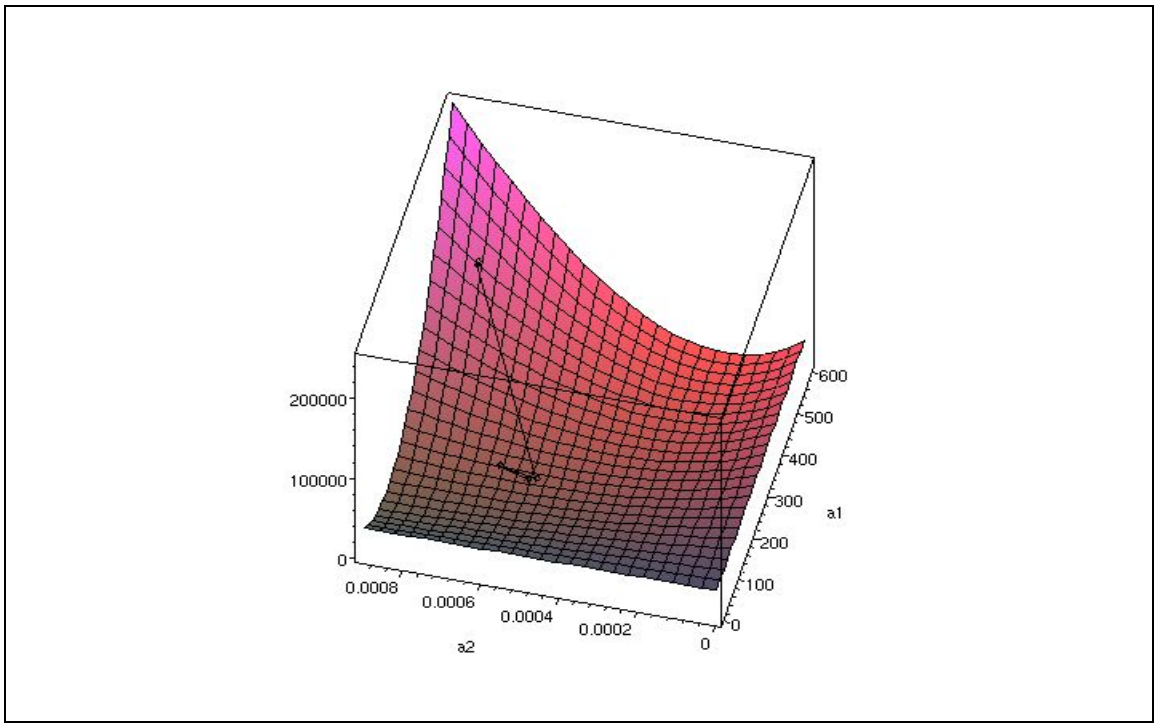


Abbildung 2.2: Die Iterationspunkte mit dem Levenberg-Marquardt-Algorithmus für Startwerte $a_1 = 500$, $a_2 = 0.0008$

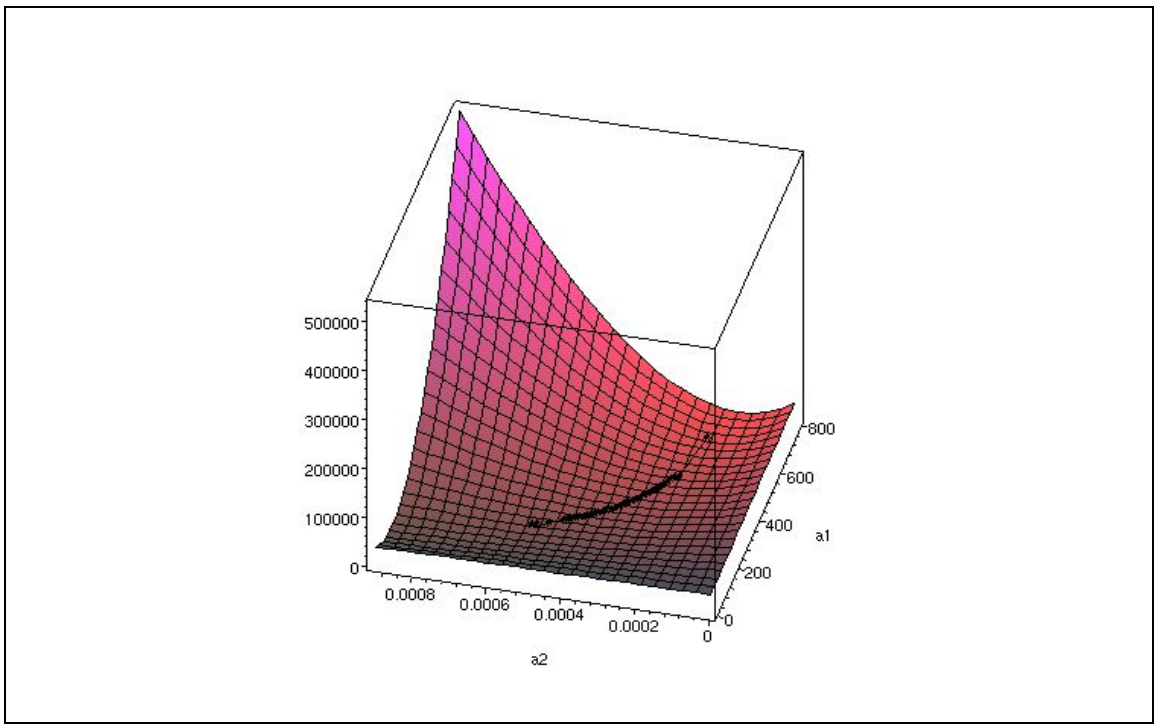


Abbildung 2.3: Die Iterationspunkte mit dem Levenberg-Marquardt-Algorithmus für Startwerte $a_1 = 500$, $a_2 = 0.0001$

Eine Anmerkung ist noch zu den Tabellen 2.2 und 2.3 zu machen. In den Tabellen mit den Iterationspunkten sind die Punkte angegeben, die der Algorithmus nacheinander berechnet. Wie man an dem Modellalgorithmus aus Kapitel 2.3.3 sieht, kann es jedoch passieren, dass sich ein Punkt $\mathbf{a}$ ergibt, der einen größeren χ^2-Wert besitzt als der vorhergehende Iterationspunkt. In diesem Fall wird die Konstante $h^{(k)}$ modifiziert und ein neuer Iterationspunkt berechnet. Nun bleibt man in diesem Fall auf dem "alten" Iterationspunkt zunächst stehen. Diese Fälle sind in der Tabelle nicht mit aufgenommen.

Wie dieses Beispiel zeigt, hängt der Erfolg des Iterationsverfahrens wesentlich von der Wahl des Startpunktes ab. In diesem einfachen Beispiel, wo die zu minimierende Funktion von nur zwei Parametern abhängt, kann die Funktion geplottet werden. Dieses Verfahren kann man jedoch in höherdimensionalen Fällen nicht mehr verwenden. Kleine Hilfen sind in diesen Fällen:

1. Um günstige Startwerte zu ermitteln, sollte auf einem vorgegebenen Gitter nach dem Parametersatz gesucht werden, für den die Funktion χ^2 minimal ist. Dieser sollte dann als Startwert für das eigentliche Verfahren verwendet werden.
2. Hat man einen Iterationszyklus beendet, kann es in einigen Fällen sinnvoll sein, die zuletzt berechneten Parameter weiter zu verwenden. Weitere Iterationen könnten letztendlich zum Minimum führen. Das wird beispielsweise bei den zweiten Startwerten aus dem obigen Beispiel deutlich. Bricht die Iteration nach z.B. vorgegebenen 20 Iterationen ab, führen weitere Iterationen mit den zuletzt berechneten Parametern zum gesuchten Minimum.

Beide Ansätze sind in dem Programmpaket zu dieser Diplomarbeit implementiert.

Dieser Datensatz mit zugehöriger Modellfunktion wurde in [7] schon eingehend untersucht. Für die Parameter können deshalb zertifizierte Werte angegeben werden. Sie liegen mit einer Genauigkeit von 6 Stellen bei $a_1 = 238.942$ und $a_2 = 0.000550$. Der Wert der Funktion χ^2 beträgt an dieser Stelle $\chi^2 = 0.126$. Mit den beiden oben angegebenen Startwerten wird das Minimum der Funktion χ^2 gefunden. Das muss jedoch nicht immer der Fall sein. Je nach Wahl der Startwerte kann ein lokales Minimum gefunden werden.

Verdeutlicht werden soll dies an der Modellfunktion

$$f : \mathbb{R}^2 \longrightarrow \mathbb{R}$$

$$f(x_1, x_2) := a_1 \sin(a_2 x_1 x_2) + a_3 \cos(a_4 x_1 + a_5 x_2) + a_6 \tag{2.25}$$

mit Startwerten $a_1 = a_2 = a_3 = a_4 = a_5 = a_6 = 1$. Die Ergebnisse des Levenberg-Marquardt-Algorithmus und des Gauß-Newton-Algorithmus sind in Tabelle 2.4 aufgelistet (gerundet auf 3 Nachkommastellen).

Parameter	Levenberg-Marquardt	modifizierter Gauß-Newton
a[1]	3.532	-2
a[2]	0.550	-4
a[3]	1.171	-3
a[4]	4.245	-2
a[5]	-1.239	4
a[6]	-0.630	-1
χ^2	69.433	$0.346 \cdot 10^{-6}$
Anzahl Iterationen	62	23

Tabelle 2.4: Berechnete Parameterwerte für beide Verfahren

Abbildung 2.4 zeigt die Datenpunkte (aus Tabelle A.1) und die Modellfunktion für die mit dem Levenberg-Marquardt-Algorithmus bestimmten Parameter, Abbildung 2.5 für die mit dem Gauß-Newton-Algorithmus bestimmten.

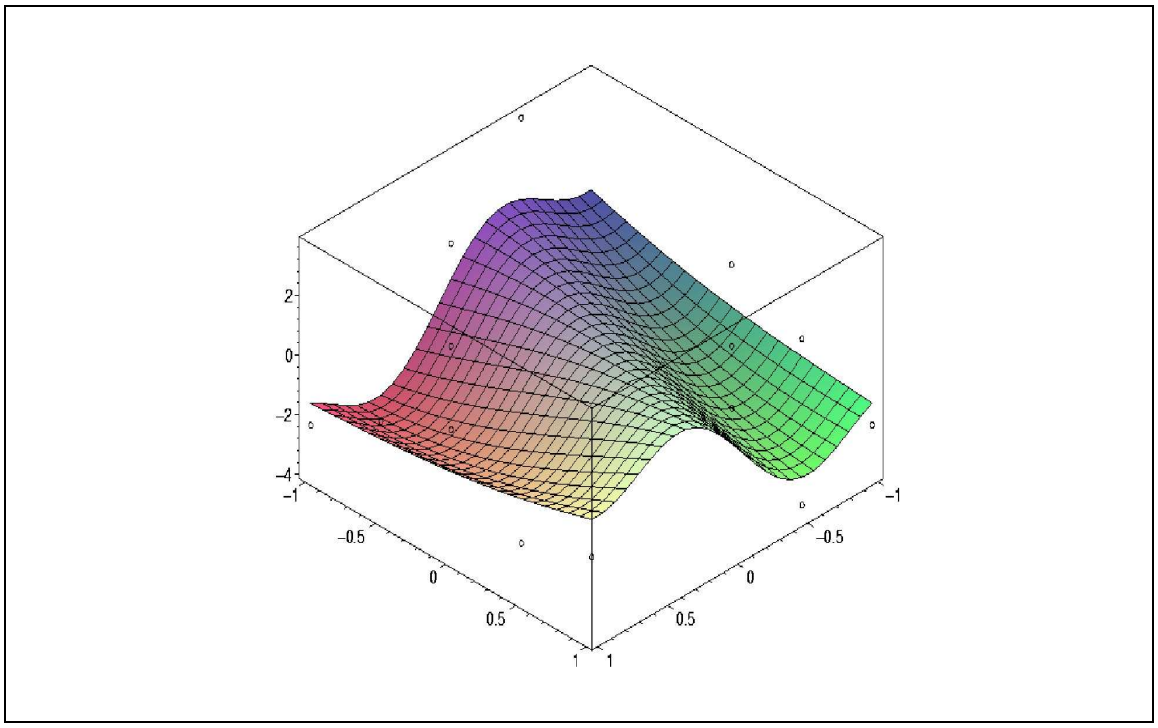


Abbildung 2.4: Die Beispielfunktion (2.25) nach Optimierung mit dem Levenberg-Marquardt-Algorithmus

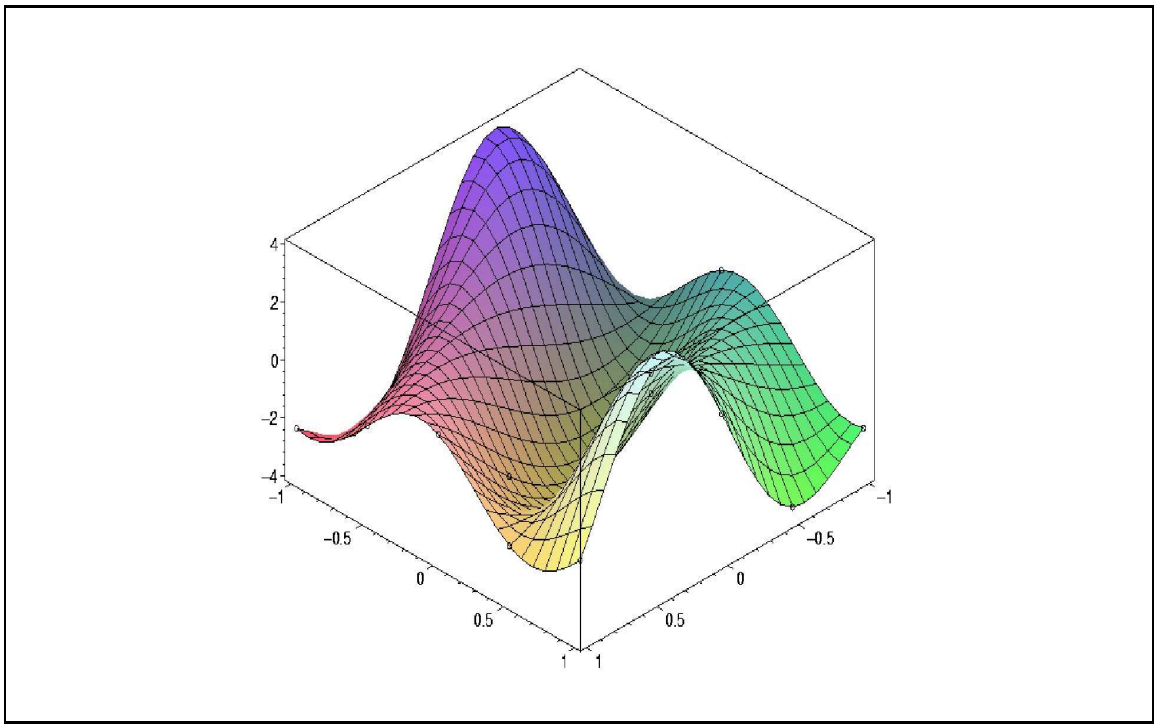


Abbildung 2.5: Die Beispielfunktion (2.25) nach Optimierung mit dem modifizierten Gauß-Newton-Verfahren

Erkennbar an Abbildung 2.4 ist, dass der Levenberg-Marquardt-Algorithmus bei der Minimierung der χ^2-Funktion zu einer lokalen Extremstelle dieser Funktion iteriert. Das ist auch aus Tabelle 2.4 erkennbar, denn der Wert der χ^2-Funktion für den Levenberg-Marquardt-Algorithmus liegt deutlich über dem Wert, den der Gauß-Newton-Algorithmus liefert.

Kapitel 3

Vergleich zweier Algorithmen zur nichtlinearen Ausgleichsrechnung

3.1 Einleitung

In diesem Kapitel sollen die in Kapitel 2 vorgestellten numerischen Algorithmen anhand einiger repräsentativer Datensätze geprüft werden. Es soll die Robustheit und Effizienz der Algorithmen getestet, sowie Vergleiche zwischen ihnen durchgeführt werden.
Als Vertreter für den Levenberg-Marquardt-Algorithmus wird dabei auf eine Implementierung aus den Numerical Recipes zurückgegriffen ([5]). Als Vertreter für den modifizierten Gauss-Newton-Algorithmus wird eine Implementierung aus der NAG Fortran 90 Library benutzt ([18]).

Die bereitgestellten Datensätze bestehen sowohl aus tatsächlich gemessenen als auch aus künstlich erzeugten Daten und stammen aus [7]. Zum Vergleich werden dort zertifizierte Werte für die Parameter angegeben. Diese Werte wurden mit mindestens zwei verschiedenen Algorithmen ermittelt, wobei mit 128-bit Genauigkeit sowie analytischen Ableitungen gearbeitet wurde.
Eingeteilt wurden die Datensätze anhand der Schwierigkeit in niedrig, mittel und hoch. Der Term "Schwierigkeit" bezieht sich dabei auf die angegebene Modellfunktion und die bereitgestellten Startwerte.
Für jeden Datensatz gibt es drei verschiedene Sätze von Startwerten. Der erste ist relativ weit vom gesuchten globalen Minimum entfernt, der zweite liegt nahe am Minimum, und der dritte sind die zertifizierten Werte selbst. Letzterer mag auf den ersten Blick als nicht sinvoll erscheinen, jedoch gibt es Algorithmen, die bei Wahl dieser Werte entweder mit einer Fehlermeldung abbrechen oder sogar vom Minimum wegiterieren.
Tabelle 3.1 gibt eine Zuordnung zwischen Datensatz und der in den nachfolgenden Abbildungen verwendeten Nummer. Zusätzlich wird noch der Schwierigkeitsgrad mit aufgeführt.

3.2 Prüfung auf Robustheit

Die folgenden Prüfungen auf Robustheit sollen aufzeigen, welches Verhalten (Konvergenz zum globalen Minimum/Konvergenz zu lokalen Minimalstellen) die hier ausgewählten Algorithmen mit den angeprochenen Datensätzen aufweisen.
Als erstes werden die Algorithmen mit den Startwerten initialisiert, die relativ weit vom globalen Minimum entfernt sind. Abbildung 3.1 zeigt die Anzahl der ausgeführten Iterationen für beide Algorithmen. Aus den ausgeführten Iterationen kann allerdings nicht auf die Effizienz der Algorithmen geschlossen werden. Eine geringere Iterationszahl des Levenberg-Marquardt-Algorithmus impliziert dabei nicht eine ebenfalls effizientere Durchführung. Auf diesen Punkt wird im nächsten Abschnitt eingegangen. Hier steht alleine das Verhalten der Algorithmen im Vordergrund.

Nummer	Name des Datensatzes	Schwierigkeitskategorie
1	Misra1a	niedrig
2	Lanczos3	niedrig
3	Gauss1	niedrig
4	Gauss2	niedrig
5	Kirby2	mittel
6	Gauss3	mittel
7	ENSO	mittel
8	Thurber	mittel
9	MGH10	hoch
10	Eckerle4	hoch
11	Benett5	hoch

Tabelle 3.1: Verwendete Datensätze

Falls ein Algorithmus mit beispielsweise 7 Iterationen terminiert, bedeutet dies allerdings nicht, dass das globale Minimum gefunden wurde. Es ist ebenfalls möglich, dass der Algorithmus zu einem lokalen Minimum konvergiert.
An der Abbildung sieht man, dass bis zum neunten Datensatz relativ wenige Iterationen benötigt werden. In allen Fällen wird auch das globale Minimum von beiden Algorithmen gefunden. Beim neunten Datensatz gab es die ersten Probleme für beide Algorithmen. Hier konnte das globale Minimum in den ersten 400 Iterationen nicht gefunden werden. Führt man die Iterationen allerdings weiter fort, so kann trotzdem mit beiden das Minimum gefunden werden. Hierfür werden um die 900 Iterationen benötigt. Beim zehnten Datensatz konvergierten beide Algorithmen gegen ein lokales Minimum.

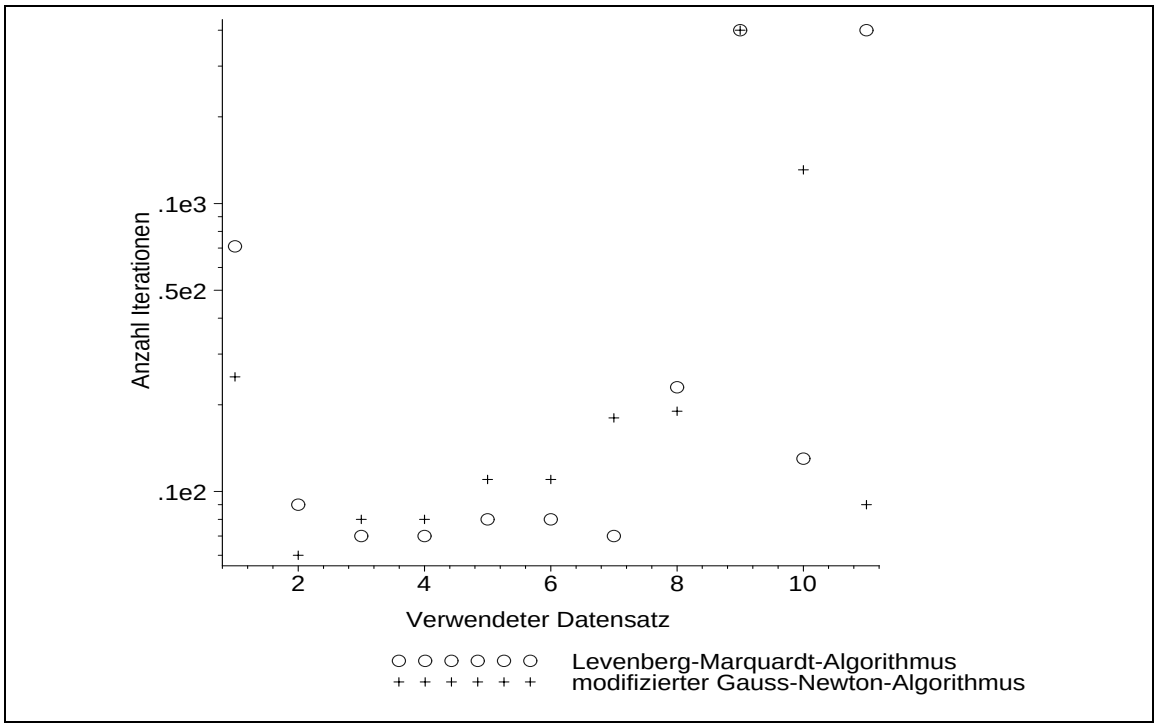


Abbildung 3.1: Anzahl Iterationen für die Startwerte, die relativ weit vom globalen Minimum entfernt sind

Als Nächstes wurden beide Algorithmen mit Startparametern versehen, die nahe am gesuchten globalen Minimum liegen. Abbildung 3.2 zeigt die Ergebnisse.
Auffällig sind auch hier die Datensätze 9 und 11. Trotz der “einfacheren” Startwerte kann mit dem Levenberg-Marquardt-Algorithmus innerhalb der vorgegebenen Iterationszahl kein befriedigendes

Ergebnis erzielt werden.

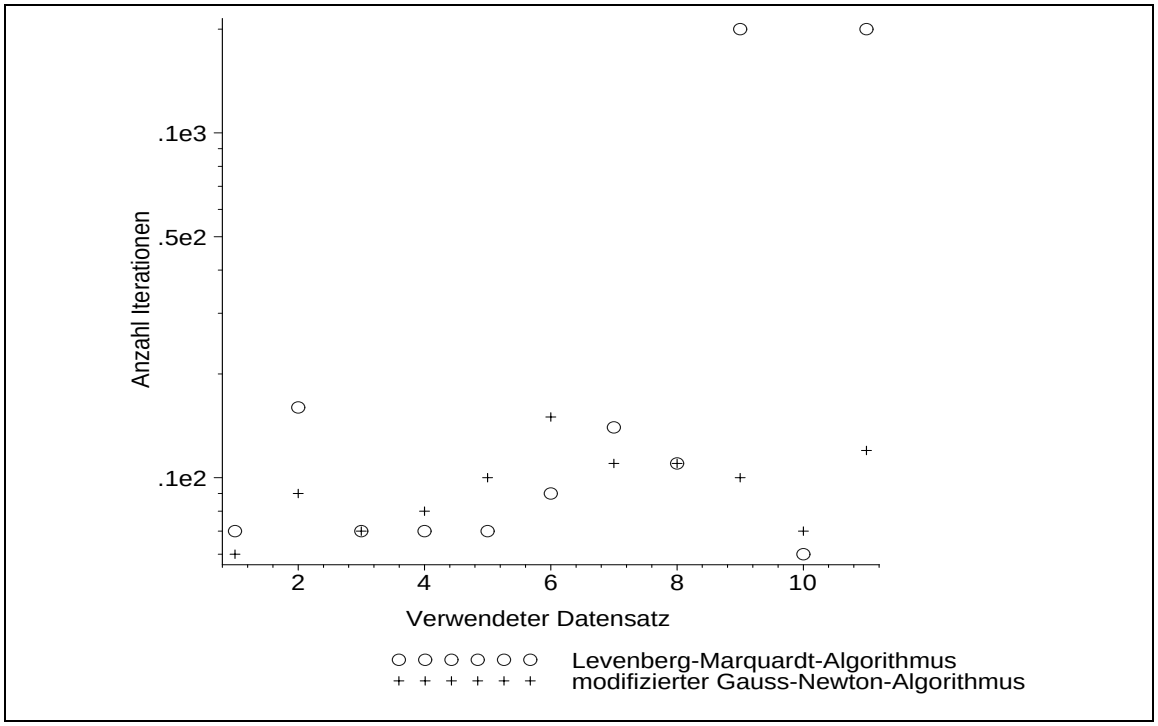


Abbildung 3.2: Anzahl Iterationen für die Startwerte, die relativ nahe am globalen Minimum liegen

Abschließend werden als Startwerte die zertifizierten Werte benutzt. Abbildung 3.3 zeigt die ermittelten Ergebnisse. Positiv fällt auf, dass keiner der Algorithmen vom gegebenen, globalen Minimum wegiteriert.

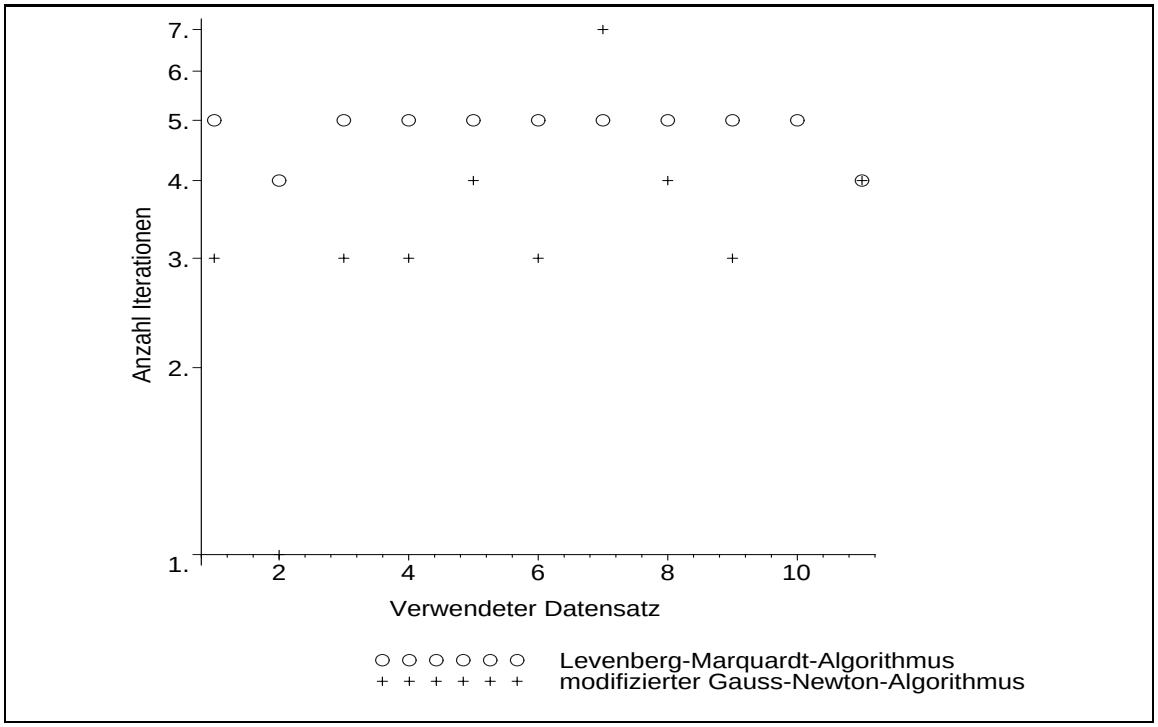


Abbildung 3.3: Anzahl Iterationen für die Startwerte, die dem globalen Minimum entsprechen

Zusammenfassend kann man sagen, dass beide Algorithmen den Tests standgehalten haben. Keiner der Algorithmen zeigte bei mehreren Datensätzen Konvergenz gegen lokale Extremstellen.

3.3 Prüfung auf Effizienz

Die nachfolgenden Tests beziehen sich auf die Effizienz der beiden Algorithmen. Hier sind mehrere Kriterien zur Messung denkbar. Zum einen könnte die Ausführungszeit der Algorithmen gemessen werden. Dieses Kriterium hat sich in der Praxis jedoch nicht bewährt, da die Laufzeiten der Algorithmen sehr gering sind (die Laufzeit liegt auf einem PC[1] für einen Datensatz mit ca. 200 Daten im Bereich von $10^{(-2)}$ Sekunden).
Eine andere Möglichkeit wäre, die Algorithmen in einer Schleife wiederholt aufzurufen und hieraus die Laufzeit für einen Datensatz zurückzurechnen. Da dieses Verfahren jedoch auch zu ungenauen Resultaten führen kann, wurde eine dritte Methode gewählt: Es wurden die benötigten Funktionsaufrufe zum Auswerten der Funktion f sowie für deren Ableitung f' gezählt. Dieses Vorgehen ist gerechtfertigt, da die Laufzeit der Algorithmen hauptsächlich dadurch bestimmt wird. Operationen wie das Lösen von linearen Gleichungssystemen fallen bei kleiner Dimension hier kaum ins Gewicht.
Eine weitere Bemerkung muss zu der Routine aus der NAG Fortran 90 Bibliothek gemacht werden: Über einen Parameter kann dort die Anzahl der Iterationen erniedrigt werden (gleichzeitig wird die Anzahl der Funktionsaufrufe erhöht), bzw. die Anzahl der Iterationen erhöht (und die Anzahl der Funktionsaufrufe erniedrigt) werden. Für die Messungen hier wurden die Default-Werte verwendet. Das Zählen der Funktionsaufrufe wird mit den gleichen Datensätzen wie im vorhergehenden Abschnitt durchgeführt.

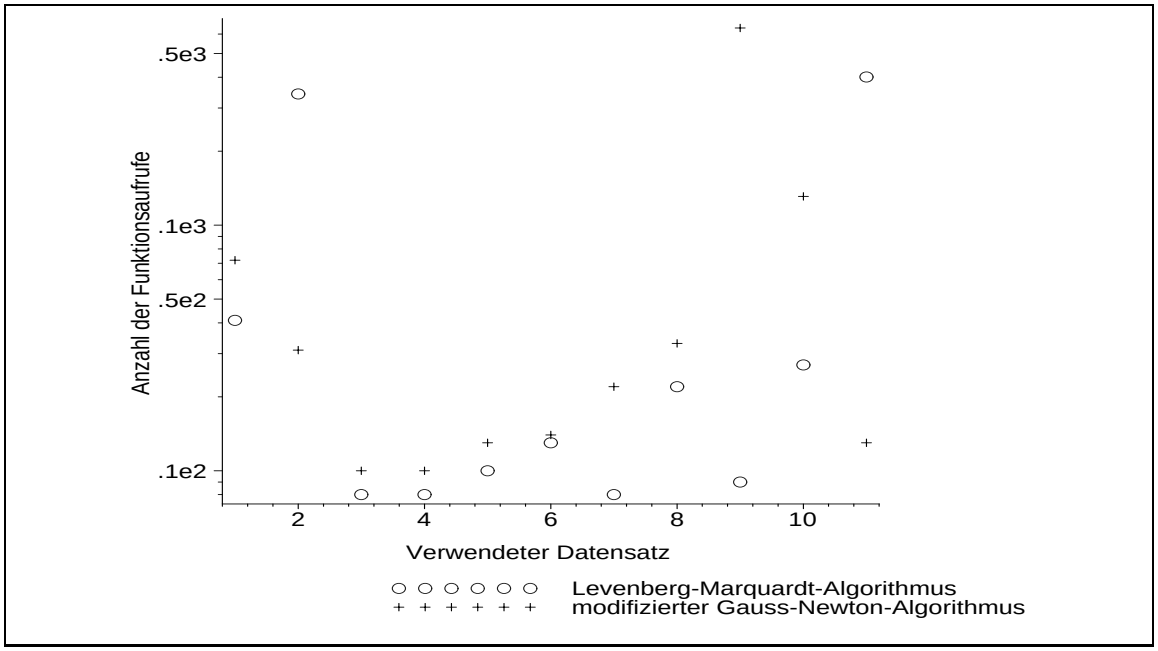


Abbildung 3.4: Anzahl Funktionsaufrufe für die Startwerte, die relativ weit vom globalen Minimum entfernt sind

Die Ergebnisse aus Abbildung 3.4 beziehen sich auf die weit vom globalen Minimum entfernten Startwerte. Auf den ersten Blick scheint der Levenberg-Marquardt-Algorithmus vor allem bei den Datensätzen 9-11 effizienter zu arbeiten. Allerdings muss hier beachtet werden, dass dieser Algorithmus in diesen Fällen nicht das globale Minimum findet, sondern zu einem lokalen Minimum konvergiert. In den restlichen Fällen benötigt der Levenberg-Marquardt-Algorithmus weniger Funktionsauswertungen, die Differenz liegt dabei jedoch nur im Bereich von 2-10 Auswertungen.

[1] Als Testsystem diente ein unter SuSE Linux 8.0 laufender AMD-Athlon PC mit 1GHz und 256 MB Hauptspeicher

Als nächstes werden die Startparameter benutzt, die näher am globalen Minimum liegen. Abbildung 3.5 zeigt die Ergebnisse. Deutliche Abweichungen ergeben sich wieder in den Datensätzen 9 und 11, bei denen der Levenberg-Marquardt-Algorithmus in den vorgegebenen 400 Iterationen das globale Minimum nicht findet. Bei den restlichen Datensätzen liegen die Verfahren im Mittel gleich auf.

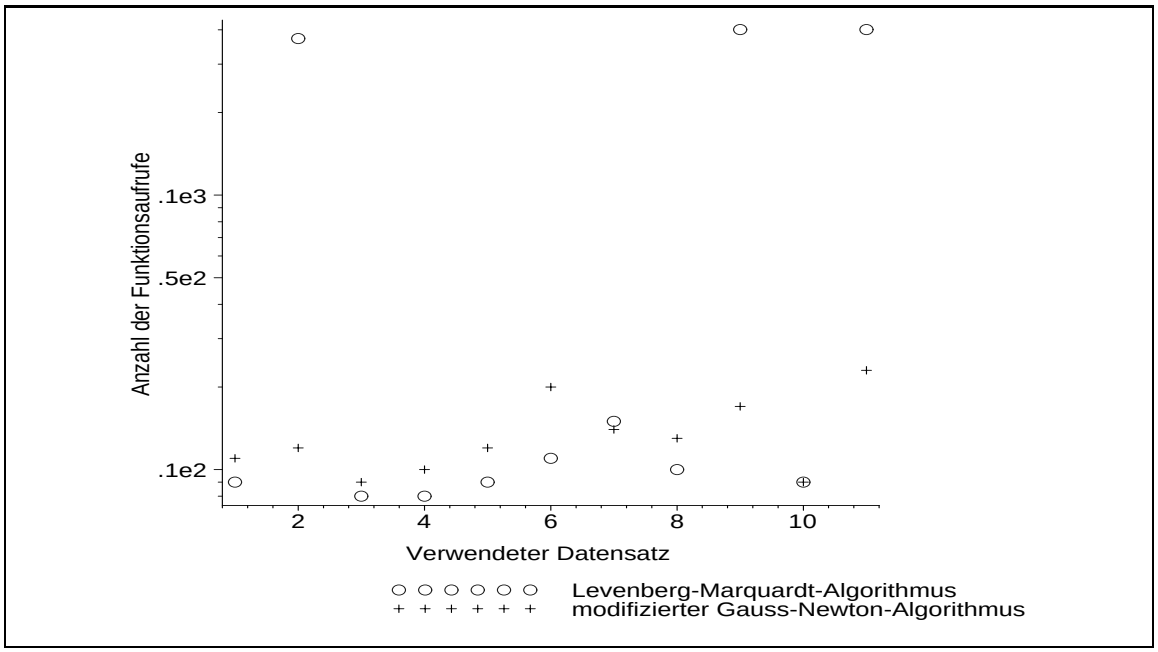


Abbildung 3.5: Anzahl Funktionsaufrufe für die Startwerte, die relativ nahe am globlen Minimum liegen

Für die Ergebnisse aus Abbildung 3.6 wurden die zertifizierten Werte als Startwerte benutzt. Hier zeigt sich, dass der modifizierte Gauss-Newton-Algorithmus im Mittel für diese Fälle weniger Funktionsauswertungen benötigt.

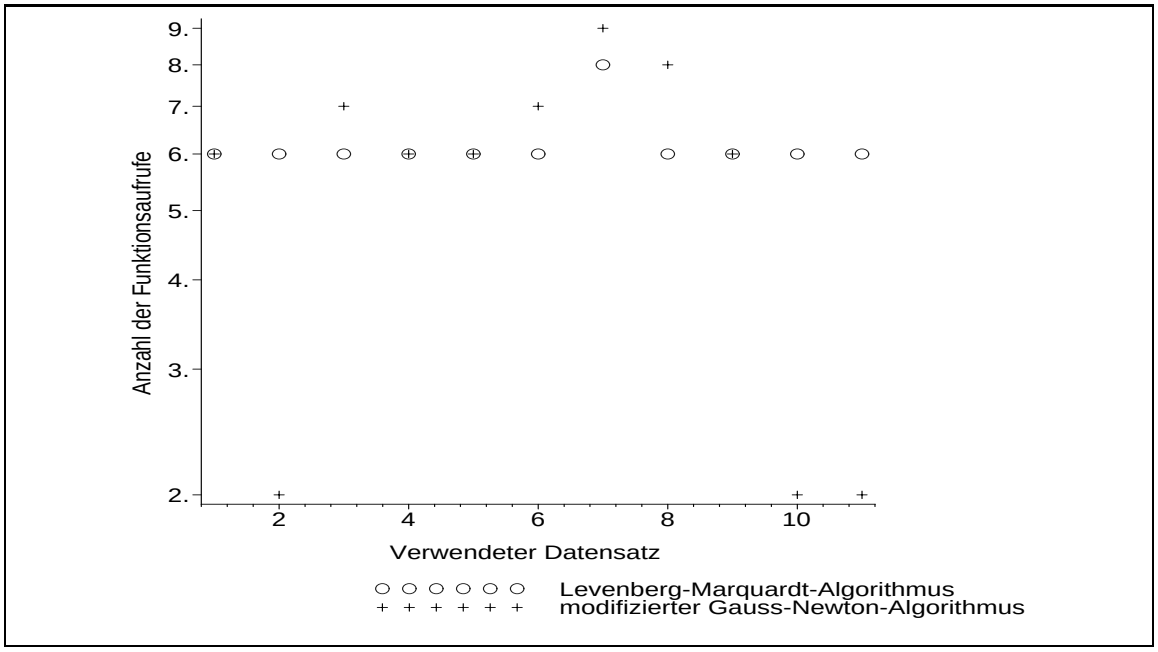


Abbildung 3.6: Anzahl Funktionsaufrufe für die Startwerte, die dem globalen Minimum entsprechen

Zusammenfassend kann gesagt werden, dass es zwischen den beiden getesteten Verfahren hinsichtlich der Effizienz keine großen Unterschiede gibt.

Kapitel 4

Programmiertechniken in Maple

4.1 Einleitung

In diesem Kapitel sollen elementare Grundlagen des Maple-Systems beschrieben werden, die bei der Erstellung des Programmpaketes wichtig waren.
Zunächst erfolgt eine Beschreibung des Prozedur- und Modulkonzepts. Das Konzept der prozeduralen Programmierung wird insbesondere in Kapitel 5 benötigt, wo die automatische bzw. symbolische Differentiation beschrieben wird. Das Modulkonzept wird zum allgemeinen Verständnis der Zusammenhänge und Abhängigkeiten des in Kapitel 6 vorgestellten Programmpaketes benötigt.
Anschließend erfolgt eine Beschreibung der Techniken, wie externe C- oder Fortran-Programme bzw. Bibliotheken eingebunden werden können. Der folgende Abschnitt zeigt die Möglichkeit, Maple-Quellcode in Programmiersprachen wie C bzw. Fortran zu übersetzen. Diese Technik wird benötigt, um symbolische, mit Maple berechnete Ableitungen auf der Ebene höherer Programmiersprachen zu nutzen. Den Abschluß des Kapitels bildet eine kurze Demonstration der Fähigkeiten des neuen Maple GUI-Tools (Maplets), das vom Programmpaket benutzt wird.

4.2 Prozeduren

In Maple können ähnlich wie in höheren Programmiersprachen sogenannte Prozeduren verwendet werden. Sie stellen einen Grundbaustein für das Programmieren mit dem Maple-System dar. Sie können sowohl zur Formulierung von Funktionen als auch zur Programmierung von komplizierten Programmen verwendet werden. Der generelle Aufbau einer Prozedur ist im Folgenden dargestellt ([14]):

```
proc( P)
  local L;
  global G;
  options O;
  description D;
  B
end proc
```

B ist eine Folge von Anweisungen, die den Körper der Prozedur bildet. P bezeichnet die formalen Parameter der Prozedur. Durch L und G können lokale bzw. globale Variable definiert werden. Auf die Bezeichner O und D, die ebenfalls wie die Deklaration von lokalen und globalen Variablen optional sind, wird nicht weiter eingegangen, da sie hier nur eine untergeordnete Rolle spielen.
Für die Definition von einfachen Funktionen (“Einzeilern”) kann eine Pfeil-Notation benutzt werden, wie es auch in der Algebra üblich ist:

```
  ( P ) -> B
```

Dies ist äquivalent zu:

```
  proc( P)
    B
  end proc
```

Es folgt ein einfaches Beispiel einer Maple-Prozedurdefinition:

```
g := proc( x::numeric)
local t;
  t := x^3;
  if( t>0) then sqrt(t) else exp(t) end if
end proc:
```

4.3 Das Modulkonzept

Die im vorigen Abschnitt beschriebenen Prozeduren ermöglichen es, mehrere Anweisungen in einem Block zusammenzufassen. In gleicher Weise erlauben es Module, mehrere Prozeduren mit zugehörigen Daten zu einem Block zusammenzufassen. Die Modulprogrammierung findet insbesondere in folgenden Bereichen Anwendung:

1. Einkapselung von Daten
2. Definition von Paketen
3. Modellierung von Objekten
4. Generische Programmierung

Der Aufbau eines Moduls sieht wie folgt aus ([10]):

```
module()
  local L;
  export E;
  global G;
  options O;
  description D;
  B
end module
```

Im Deklarationsblock sind alle Argumente optional. Um auf exportierte Elemente des Moduls zugreifen zu können, dient der “member selection operator” :- . Es folgt ein einfaches Beispiel:

```
m1 := module()
  export e1,e2;
  local a;
  a:=2;
  e1:=x->a^x;
  e2:=x->sin(a*x);
end module:
```

Ein wesentlicher Unterschied zur Prozedurdefinition ist der Export von lokalen Variablen, d.h. diese Variablen sind nach Ausführung der Moduldefinition außerhalb des Moduls bekannt.

Auf `e1` kann nun durch

```
m1:-e1(2);
```

$$4$$

zugegriffen werden. Andere Möglichkeiten, auf die exportierten Elemente des Moduls zuzugreifen, bieten die `use`-Umgebung oder das `with`-Kommando.

4.4 Einbindung externer Routinen

4.4.1 Einbindung externer C-/Fortran-Bibliotheksprogramme

Mit Maple ist es möglich, externe Fortran- oder C-Routinen zu benutzen. Als Basis für die hier beschriebenen Beispiele dient jeweils eine shared Library. Die Erstellung dieser Library ist plattformabhängig, kann jedoch unter der Berücksichtigung des Betriebssystems direkt in Maple erfolgen, da mit dem `system`-Befehl Kommandos an das Betriebssystem abgesetzt werden können. Die folgenden Beispiele beziehen sich auf ein Linux-System mit den folgenden Compilern:

Fortran-Compiler: f90 (NAGWare Compiler, bietet eine gute Diagnostik)

C-Compiler: gcc (GNU C/C++ Compiler)

Damit Maple die erstellten Libraries findet, muss vor dem Start des Worksheet-Interfaces die Shell-Variable LD_LIBRARY_PATH erweitert werden. Dies kann auf der Shell-Ebene (als Referenzshell wurde hier die bash - Bourne-Again SHell - verwendet) erfolgen, z.B. mit

```
LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/Verzeichnis_des_externen_Programms
```

oder durch Erweiterung der entsprechenden Variablen im Shell-Skript maple.

Wie im Nachfolgenden gezeigt wird, lässt sich die Programmausführung durch die Nutzung von übersetzten Bibliotheksprogrammen anstatt Maple-Code deutlich beschleunigen.

Einbinden einer externen C-Funktion

Das Einbinden von Programmen aus der NAG-C-Bibliothek wird anhand der Funktion `f04arc` erläutert, mit der lineare Gleichungssysteme gelöst werden können. Zunächst wird das shared object file erstellt:

```
system("gcc -o linsol_c.so -shared -fpic -I/usr/local/include
        linsol_c.c -lnagc");
```

$$0$$

Der C-Quellcode (doppelte Genauigkeit) zum Aufruf des Bibliotheksprogramms befindet sich in der Datei `linsol_c.c`.

```
#include <nag.h>
#include <nag_types.h>
#include <stdio.h>
#include <nag_stdlib.h>
#include <nagf04.h>

int linsol( int dim, double a[dim][dim], double b[dim], double
sol[dim])
{
static NagError fail;

  f04arc( dim, (double*) a, dim, b, sol, &fail);
```

```
  return fail.code;
}
```

Definition der Maple-Prozedur mit Link zur C-Funktion:

```
> mylinsol := define_external( 'linsol',
                               'dim'   :: integer[4],
                               'mat'   :: ARRAY(1..n*n, float[8]),
                               'rhs'   :: ARRAY(1..n, float[8]),
                               'sol'   :: ARRAY(1..n, float[8]),
                               'RETURN'::integer[4],
                               'LIB'="linsol_c.so");
```

$mylinsol :=$ **proc**(dim::$integer_4$, mat::rtable($datatype = float_8$),
rhs::rtable($datatype = float_8$), sol::rtable($datatype = float_8$))
option $call\_external$;
call_external(Array(1..17, [...], datatype = integer[4], storage = rectangular\
, order = Fortran_order, readonly), args)
end proc

Aufruf der externen C-Funktion und Vergleich mit der vom Maple-System berechneten Lösung:

```
> with( LinearAlgebra):
  dim := 5:
  A := RandomMatrix( dim, dim, outputoptions=[datatype=float[8],
  order=C_order]);
  b := RandomVector( dim, outputoptions=[datatype=float[8],
  order=C_order]);

  Maple_Loesung=LinearSolve(A,Transpose(b));
  sol:=Vector(dim,datatype=float[8],order=C_order):
  mylinsol(dim,A,b,sol):
  f04arc_Loesung=Transpose(sol);
```

$$A := \begin{bmatrix} 73. & -30. & -13. & -72. & 45. \\ -26. & 70. & -88. & -33. & -62. \\ 92. & -98. & -1. & -6. & -41. \\ 20. & -37. & -85. & -96. & 76. \\ -51. & 46. & -8. & 9. & 93. \end{bmatrix}$$

$$b := \begin{bmatrix} -25. \\ 71. \\ -57. \\ -56. \\ 26. \end{bmatrix}$$

$Maple\_Loesung = [0.570020380021052486,\ 1.09854992237056904,$
$-0.273285361205218957,\ 0.502201349254120077,\ -0.0233156137731325569]$

$f04arc\_Loesung = [0.570020380021052597,\ 1.09854992237056904,$
$-0.273285361205219179,\ 0.502201349254120299,\ -0.0233156137731325604]$

Das Einbinden von externen Fortran-Programmen ist auf die gleiche Art und Weise möglich, daher soll es nur noch an einem kleineren Beispiel gezeigt werden.

Einbinden einer externen Fortran-Funktion

Das Einbinden von Programmen aus einer Fortran-Bibliothek soll hier beispielhaft an der Funktion `arctanh(x)` aus der NAG-Fortran90-Bibliothek (siehe Modul 3.1, nag_inv_hyp_fun, in [18]) vorgeführt werden. Zunächst wird das shared object file erstellt:

```
> sourcepath:="./kap10":
> currentdir();
```

"/usr/users/kuelheim/Studium/FH-Juelich/skripte/ComputerMathe"

```
> currentdir(sourcepath);
> system("f90 -o arctanh.so arctanh.f90 -fpic -shared
          -I/usr/local/nagfl90/nag_mod_dir
          /usr/local/nagfl90/LOCALlib/libnagfl90.a");
```

$$0$$

Bei Anwendung des Shell-Skripts nagfl90 können die Pfadangaben für die NAG fl90 Library entfallen:

```
> system("nagfl90 -o arctanh.so -shared -fpic arctanh.f90");
```

$$0$$

```
> printcode("arctanh.f90"):

module global_constan
  implicit none
  private
    integer, parameter, public :: wp =
      selected_real_kind(2*precision(1.0))

end module global_constant

function arctanh( x) result(y)
use nag_inv_hyp_fun, only : nag_arctanh
use global_constant, only : wp

  implicit none
  real(wp), intent(in) :: x
  real(wp)             :: y

    y = nag_arctanh( x )
end function arctanh
```

Definition der Maple-Prozedur mit Link zum externen Fortran-Programm im shared object file:

```
> myarctanh:=define_external('arctanh',
                             'FORTRAN',
                             'a'::float[8],
                             'RETURN'::float[8],
                             'LIB'="arctanh.so"):
```

Aufruf der generierten Maple-Funktion:

```
> x:=myarctanh(0.9);
```

$$x := 1.47221948958322036$$

4.4.2 Performancevergleich

Für einen Performance-Vergleich zwischen Maple-Code und übersetztem Fortran-Programm werden die Bessel-Funktionen der ersten Art, `BesselJ`, von Ordnung 0 bis 500 berechnet:

```
> t0:=time():
  seq(BesselJ(n,500.),n=0..500):
  time_maplecode:=time()-t0;
```

$$time\_maplecode := 14.111$$

```
> system("f90 -o besselj.so besselj.f90 -shared -fpic
         -I/usr/local/nagfl90/nag_mod_dir
         /usr/local/nagfl90/LOCALlib/libnagfl90.a"):

> BesselJ_nag:=define_external('bessel_j_nag',
                               'FORTRAN',
                               'nu'::float[8],
                               'z'::float[8],
                               'RETURN'::float[8],
                               'LIB'="besselj.so");
```

$BesselJ\_nag :=$ **proc**(ν::{$name$, $numeric$}, z::{$name$, $numeric$})
option $call\_external$;
call_external(Array(1..26, [...], datatype = integer[4], storage = rectangular\
, order = Fortran_order, readonly), args)
end proc

```
> t0:=time():
  seq(BesselJ_nag(n,500.),n=0..500):
  time_externalcode:=time()-t0;
```

$$time\_externalcode := 0.030$$

```
> time_maplecode/time_externalcode;
```

$$470.3666667$$

Bei der Benutzung der Bibliotheksfunktion verringert sich die Ausführungszeit um mehr als zwei Größenordnungen.
Zunehmend werden NAG Routinen auch innerhalb von Maple genutzt, z.B. im Paket `LinearAlgebra`, wenn Funktionen mit der Option `datatype=float[8]` aufgerufen werden.

4.5 Generierung von Fortran- und C-Code

Das Maple-System bietet die Möglichkeit, Maple-Quellcode in andere Sprachen zu übersetzen. Hierzu zählen die Sprachen C, Fortran und Java. Im Folgenden wird sich auf die beiden zuerst erwähnten beschränkt ([10]). Gegeben sei eine Liste mit Datenpunkten:

```
> p:=[[-3/2,-1/2],[1,1],[2,4],[3,3]];
```

$$p := [[\frac{-3}{2}, \frac{-1}{2}], [1, 1], [2, 4], [3, 3]]$$

Mit der Prozedur `Spline` aus dem Paket `CurveFitting` wird die zugehörige kubische Spline-Funktion berechnet.

```
> with(CurveFitting);
> f:=Spline(p,x);
```

$$f := \begin{cases} -\frac{182}{225} + \frac{473}{675}\,x + \frac{68}{75}\,x^2 + \frac{136}{675}\,x^3 & \text{x< } 1 \\ \frac{46}{45} - \frac{647}{135}\,x + \frac{32}{5}\,x^2 - \frac{44}{27}\,x^3 & \text{x< } 2 \\ -\frac{946}{45} + \frac{3817}{135}\,x - \frac{152}{15}\,x^2 + \frac{152}{135}\,x^3 & otherwise \end{cases}$$

Den Graphen dieser stückweise definierten Funktion zeigt Abbildung 4.1.

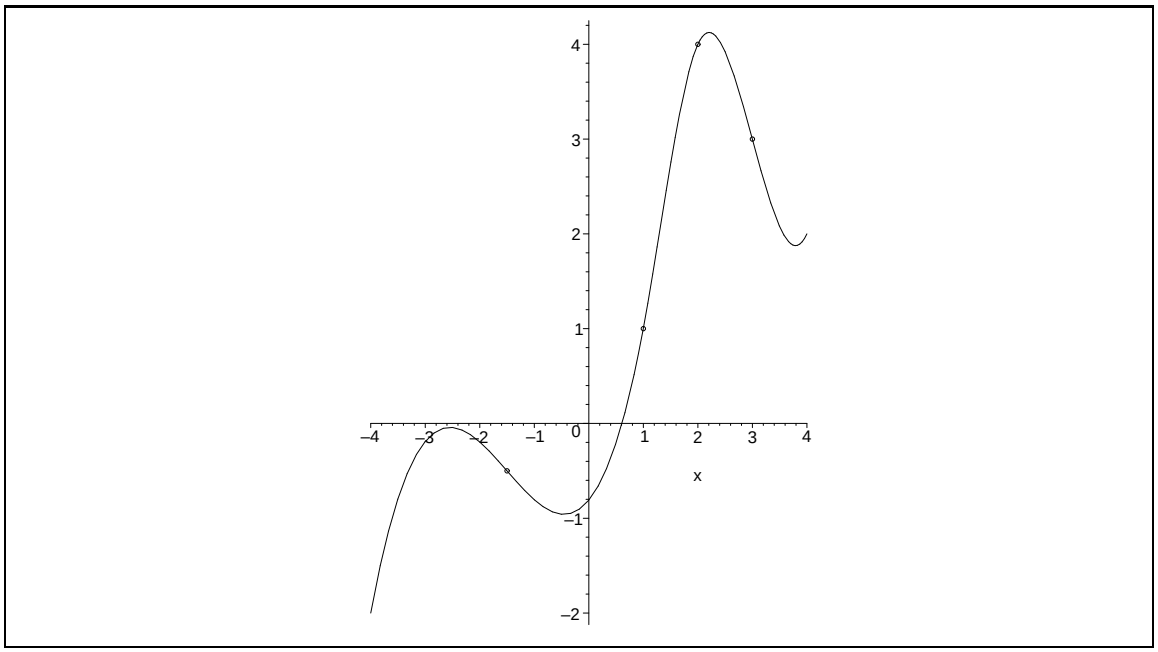


Abbildung 4.1: Spline Interpolation

Damit eine Übersetzung nach Fortran, C oder Java erfolgen kann, muss zunächst beachtet werden, dass die `Spline`-Funktion als Ergebnis eine `piecewise`-Funktion liefert. Diese Funktion muss in eine Prozedur mit "if statements" transformiert werden.

```
> f:=codegen[prep2trans](f);
```

$f := \mathbf{proc}(x)$
option $operator,\ arrow$;
  if $x < 1$ **then** $-182/225 + 473/675 * x + 68/75 * x^2 + 136/675 * x^3$
  elif $x < 2$ **then** $46/45 - 647/135 * x + 32/5 * x^2 - 44/27 * x^3$
  else $-946/45 + 3817/135 * x - 152/15 * x^2 + 152/135 * x^3$
  end if
end proc

Die eigentliche Übersetzung erfolgt schließlich mit den Funktionen `Fortran` bzw. `C` aus dem Paket `CodeGeneration`.

```
> with(CodeGeneration);
```

$$[C,\ Fortran,\ Java]$$

```
> Fortran(f);
```

```
      doubleprecision function f (x)
      doubleprecision x
      doubleprecision cgret

      if (x .lt. 1)
        cgret = -0.182D3 / 0.225D3 + 0.473D3 / 0.675D3 * dble(x) + 0.68
     #D2 / 0.75D2 * dble(x ** 2) + 0.136D3 / 0.675D3 * dble(x ** 3)
      else if (x .lt. 2)
        cgret = 0.46D2 / 0.45D2 - 0.647D3 / 0.135D3 * dble(x) + 0.32D2
     #/ 0.5D1 * dble(x ** 2) - 0.44D2 / 0.27D2 * dble(x ** 3)
      else
        cgret = -0.946D3 / 0.45D2 + 0.3817D4 / 0.135D3 * dble(x) - 0.15
     #2D3 / 0.15D2 * dble(x ** 2) + 0.152D3 / 0.135D3 * dble(x ** 3)
      end if
      f = cgret
      return
      end
```

```
> C(f);
```

```
#include <math.h>
double f (int x)
{
  double cgret;
  if (x < 1)
    cgret = -0.182e3 / 0.225e3 + 0.473e3 / 0.675e3 * (double) x +
0.68e2 / 0.75e2 * (double) x * (double) x + 0.136e3 / 0.675e3 *
(double) (int) pow((double) x, (double) 3);
  else if (x < 2)
    cgret = 0.46e2 / 0.45e2 - 0.647e3 / 0.135e3 * (double) x + 0.32e2
/ 0.5e1 * (double) x * (double) x - 0.44e2 / 0.27e2 * (double) (int)
pow((double) x, (double) 3);
  else
    cgret = -0.946e3 / 0.45e2 + 0.3817e4 / 0.135e3 * (double) x -
0.152e3 / 0.15e2 * (double) x * (double) x + 0.152e3 / 0.135e3 *
(double) (int) pow((double) x, (double) 3);
  return(cgret);
}
```

4.6 Maplets

4.6.1 Einführung

Das Paket **`Maplets`** bietet eine Programmierumgebung zur Erstellung eigener graphischer Benutzeroberfächen für Maple-Anwendungen. Maplets sind kein Ersatz für Worksheets, sondern ergänzen diese. Da die Programmierung graphischer Oberflächen in Maple neu ist, soll ein kurzer Überblick über deren Leistungsvermögen gegeben werden.

Als erstes Maplet-Beispiel soll die graphische Benutzeroberfläche für das `CurveFitting`-Paket dienen.

```
> CurveFitting[Interactive]([[1,2],[3,5],[4,7],[7,8]]);
```

In der `CurveFitting[Interactive]`-Prozedur wird ein Aufruf von `Maplets[Display]` ausgeführt. Diese Routine sendet eine Maplet-Definition an den Maple-Kernel, welcher daraufhin eine *Java Virtual Machine(JVM)* startet, an die die Maplet-Definition übergeben wird. Diese Definition wird nun in verschiedene *Java* Klassen konvertiert, die dann auf dem Bildschirm ausgegeben werden. Zwischen dem Maple-Kernel und dem Maplet ist Kommunikation möglich. So können dann vom Maplet aus Maple-Kommandos ausgeführt werden. Ebenso können Werte des Maplets abgefragt und aktualisiert werden.

Weitere Maplets in Maple sind der interactive plot builder sowie das unter dem File-Menü in der Worksheet-Oberfläche angebotene Menü für die Einstellung von “file preferences”.

4.6.2 Beispiele für Maplets

Im Folgenden sollen zwei einfache Beispiele für Maplets angegeben werden. Abbildung 4.2 zeigt die Möglichkeiten, über Textfelder Eingaben in das Maplet zu tätigen. Über Buttons kann ebenfalls auf das Maplet eingewirkt werden. Generell kann man sagen, dass mit den Maplets die gleichen graphischen Elemente wie in Java erzeugt werden können, da jedes Maplet-Element zu einer Java-Klasse gehört. Zudem kann vom Maplet aus auf die Maple-Kernelfunktionen zurückgegriffen werden. So sind beispielsweise Operationen wie Integrieren und Differenzieren vom Maplet aus ausführbar. Diese Tatsache nutzt dieses Maplet, um eine vom Benutzer übergebene Funktion nach x zu differenzieren. Das Resultat kann dabei ebenfalls wieder im Maplet dargestellt werden.

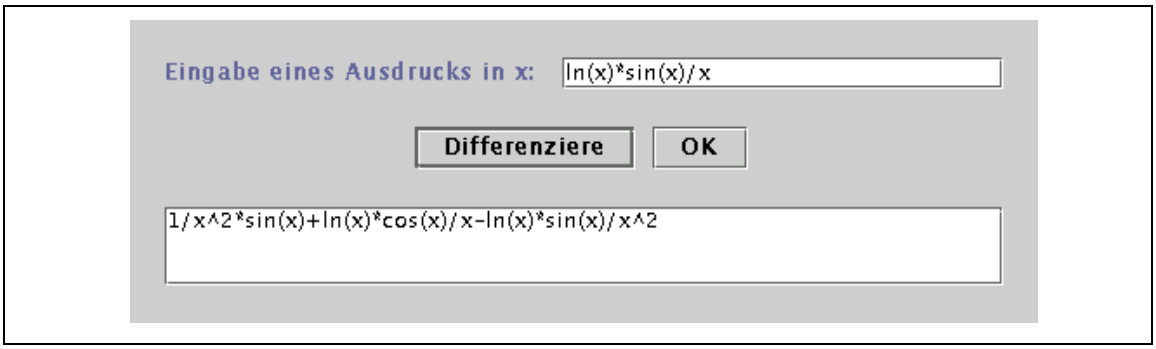


Abbildung 4.2: Interaktionen zwischen dem Benutzer und dem Maple-Kernel

Das nächste Beispiel zeigt die Möglichkeit, in Maplets graphische Ausgaben darzustellen. Abbildung 4.3 enthält das Bild eines Maplets zur graphischen Darstellung einer vom Benutzer angegebenen Funtion in x.

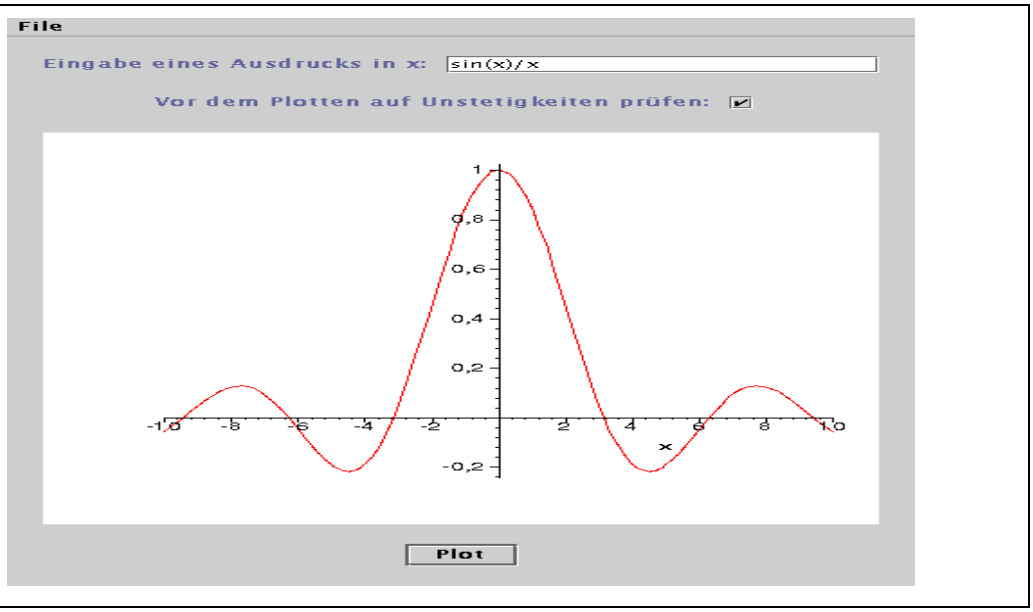


Abbildung 4.3: Plotausgaben im Maplet

Weitere Informationen zu Maplets finden sich in [10] und [13].

Kapitel 5

Methoden der Differentiation

5.1 Einleitung

In diesem Kapitel werden zwei Verfahren vorgestellt, die die Ableitung einer Funktion bzw. eines Programms berechnen können. Das bekanntere von beiden ist sicher die symbolische Differentiation. Eine Alternative dazu stellt die automatische Differentiation dar. Mit ihr können nicht nur einfache Funktionsausdrücke differenziert werden, sondern auch komplexe Programme. In dieser Arbeit werden Funktionen der Art

$$f : \mathbb{R}^n \longrightarrow \mathbb{R}.$$

und ihre Ableitungen betrachtet. Das impliziert, dass die Funktion f auf ihrem Definitionsbereich differenzierbar ist. Zusätzlich soll f nur aus einfachen, elementaren Funktionen zusammengesetzt sein.

Jede Funktion dieser Art kann als Programm geschrieben werden. Verdeutlicht werden soll das am Beispiel der Funktion

$$f : \mathbb{R}^2 \longrightarrow \mathbb{R}$$

$$f(x_1, x_2) := x_1 \cdot x_2 + \sin(x_1) + \exp(x_1 - x_2). \tag{5.1}$$

Diese kann, wie der nachfolgende Pseudocode zeigt, zerlegt werden in:

```
program f(x1,x2)
  t0 = x1 * x2
  t1 = sin(x1)
  t2 = x1-x2
  t3 = exp(t2)
  t4 = t0+t1
  t5 = t4+t3
return t5
```

Compiler machen prinzipiell das Gleiche, wenn es darum geht, einen komplexen mathematischen Ausdruck auszuwerten. Der Ausdruck wird so lange zerlegt, bis nur noch elementare Ausdrücke übrigbleiben. Diese werden berechnet und anschließend wieder zum Ausgangsausdruck zusammengesetzt.

5.2 Symbolische Differentiation

Die symbolische Differentiation ist die klassische Differentiationsmethode. Hierbei werden mit Hilfe der bekannten Differentiationsregeln der Analysis die Ableitungen (bzw. partiellen Ableitungen)

der Funktion berechnet. Das Resultat der symbolischen Differentiation ist wieder eine Funktion, die die geforderte Berechnungssequenz als Formel bereitstellt.
Sei f wie in (5.1) vorgegeben.
Im Computeralgebrasystem Maple ist die symbolische Differentiation mit dem Operator `diff` möglich. Die symbolischen partiellen Ableitungen nach x_1 bzw. x_2 ergeben sich hiermit als:

```
> diff( f(x1,x2), x1);
```

$$x2 + \cos(x1) + e^{(x1-x2)}$$

```
> diff( f(x1,x2), x2);
```

$$x1 - e^{(x1-x2)}$$

5.3 Automatische Differentiation

Automatische Differentation (AD), auch algorithmische Differentiation oder Differentiation von Algorithmen genannt, beschäftigt sich mit dem Problem, zu einem in algorithmischer Form vorliegenden Programm P ein Programm p zu bestimmen, welches die Ableitung von P berechnet. Das Programm P hat dabei die Eigenschaft, numerische Werte als Eingabeparameter zu erhalten und numerische Werte als Ausgabeparameter zurückzugeben. Die Auswertung von p sollte dabei mit der gleichen Exaktheit und mit der gleichen Effizienz erfolgen wie die Auswertung von P. Automatische Differentiation erzeugt also eine transformierte Version des Programms P ([8], [9], [10]). Die automatische Differentiation wird hier zur Berechnung der Ableitungen der Funktion χ^2 verwendet, die in den Algorithmen aus Kapitel 2 benötigt wird.

Zunächst einige Abgrenzungen, um zu zeigen, was automatische Differentiation nicht ist.

Sei $f : \mathbb{R} \longrightarrow \mathbb{R}$ stetig differenzierbar. Ein sehr einfaches Verfahren, um numerische Näherungen für die Ableitung von f an der Stelle $x_0 \in \mathbb{R}$ zu erhalten, ist der Differenzenquotient

$$f^{'}(x_0) \approx \frac{f(x_0 + h) - f(x_0)}{h}$$

oder

$$f^{'}(x_0) \approx \frac{f(x_0 + h) - f(x_0 - h)}{2h}.$$

Diese numerische Annäherung der Ableitung ist zu vermeiden, da es für kleine h zu Auslöschungen in der Differenzbildung kommen kann, die die signifikanten Stellen des Ergebnisses deutlich verringern. Mit diesen Verfahren erhält man insgesamt nur Werte, die ungefähr $\frac{1}{3}$ bis $\frac{2}{3}$ der tatsächlich signifikanten Stellen wiedergeben ([8]).

Ebenso unterscheidet sich automatische Differentiation von der symbolischen Differentiation. Symbolische Differentiation produziert “formelmäßige” Ableitungen eines Ausdrucks.

Sei $f : \mathbb{R}^2 \longrightarrow \mathbb{R}$ mit

$$f(x, y) := \sin(2x + y) + \cos(x) \exp(y^2)$$

gegeben, dann ergeben sich durch symbolische Differentiation die Ausdrücke

$$\frac{\partial f(x, y)}{\partial x} = 2\cos(2x + y) - \sin(x) \exp(y^2)$$

$$\frac{\partial f(x, y)}{\partial y} = cos(2x + y) + 2y \sin(x) \exp(y^2)$$

Mögliche Probleme sollen an einem Beispiel verdeutlicht werden.
Sei $f : \mathbb{R}^n \longrightarrow \mathbb{R}$ mit

$$f(\mathbf{x}) := \prod_{i=1}^{n} x_i = x_1 \cdot x_2 \cdot \cdots \cdot x_n$$

Bildet man von dieser Funktion den Gradienten, so ergibt sich:

$$\bigtriangledown f(\mathbf{x}) = \begin{pmatrix} x_2 \cdot x_3 \cdot \ldots \cdot x_i \cdot x_{i+1} \cdot \ldots \cdot x_n \\ x_1 \cdot x_3 \cdot \ldots \cdot x_i \cdot x_{i+1} \cdot \ldots \cdot x_n \\ \vdots \\ x_1 \cdot x_2 \cdot \ldots \cdot x_i \cdot x_{i+1} \cdot \ldots \cdot x_{n-1} \end{pmatrix}$$

Man erkennt, dass der komplette Ausdruck des Gradienten für große n viel Speicherplatz benötigt. Ebenso befinden sich in den einzelnen Komponenten des Gradienten viele Ausdrücke, die nur einmal hätten berechnet werden müssen. Mit Hilfe der symbolischen Differentiation ist der oben angegebene Ausdruck erzeugbar. Die automatische Differentiation geht nun nicht diesen langen Weg über die symbolischen Ergebnisse. Eine Schlüsselrolle spielen hier die Zwischenergebnisse, wie z.B. die partiellen Produkte $x_1 \cdot x_2 \cdot \cdots \cdot x_i$, die ebenfalls bei der Berechnung der Ausgangsfunktion f berechnet werden müssen. Diesen Zwischenergebnissen werden separate Namen zugewiesen, auf die im weiteren Verlauf wieder zugegriffen werden kann, speziell bei der Berechnung des Gradienten bzw. höherer Ableitungen.

Automatische Differentiation kann weit mehr als hier beschrieben werden kann. Hier soll sie nur dazu verwendet werden, von einem gegebenen Programm P, das aus einer Kombination von Elementarfunktionen besteht, den Gradienten zu berechnen. Bezeichnet Φ die Menge der Elementarfunktionen, also $\Phi \supset \{\sin, \cos, \exp, \sqrt{}, ...\}$, dann werden hier Programme P betrachtet, die aus einer Kombination von Elementarfunktionen $v_1, ..., v_m \in \Phi$ bestehen.
Mit Hilfe von AD kann man zudem auch komplexe Programme ableiten. In Maple steht die automatische Differentiation mittels des Operators `D` zur Verfügung, und soll nun an einem komplexeren Beispiel veranschaulicht werden. Zunächst wird die Prozedur angegeben, deren Ableitung mit Hilfe der automatischen Differentiation bestimmt werden soll:

```
> g := proc(x)
    local i, result;

    result := 0;
    if x > 0 then
      result := 15/2*x+5;
    else
      for i from 1 to 5 do
        result := result + sqrt(exp(x*i));
      end do:
    end if;
    result;
  end proc;
```

$g := \mathbf{proc}(x)$
local $i,\ result$;
$result := 0$;
if $0 < x$ **then** $result := 15/2 * x + 5$
else for i **to** 5 **do** $result := result + \mathrm{sqrt}(\exp(x * i))$ **end do**
end if;
$result$
end proc

Die durch das Programm definierte Funktion $g|_{\mathbb{R}\setminus\{0\}}$ ist auf ihrem Definitionsbereich zunächst einmal differenzierbar. Dass das auch für den Punkt 0 gilt, ergibt sich aus

```
> Limit( ('g(h)'-g(0))/h,h=0,left);
```

$$\lim_{h \to 0-} \frac{\mathrm{g}(h) - 5}{h}$$

```
> evalf( '%');
```

$$7.500000000$$

und der Tatsache, dass der Wert des rechtsseitigen Grenzwertes offensichtlich ebenfalls 7.5 beträgt. Nun kann die Ableitung des Programms mittels automatischer Differenzierung berechnet werden:

```
> g_strich := D(g);
```

$g\_strich := \mathbf{proc}(x)$
local $i,\ resultx,\ result$;
$resultx := 0$;
$result := 0$;
if $0 < x$ **then** $resultx := 15/2$; $result := 15/2 * x + 5$
elsefor i **to** 5 **do**
$resultx := resultx + 1/2 * i * \exp(x * i)/\mathrm{sqrt}(\exp(x * i))$; $result := result + \mathrm{sqrt}(\exp(x * i))$
end do
end if;
$resultx$
end proc

In Abbildung 5.1 wird g und g_strich dargestellt. Zusammengefasst kann man also sagen, dass das Ergebnis der Anwendung von automatischer Differentiation auf ein Programm P ein Programm p ist, dass dessen Ableitung für einen bestimmten Wert berechnet und keine Formel.

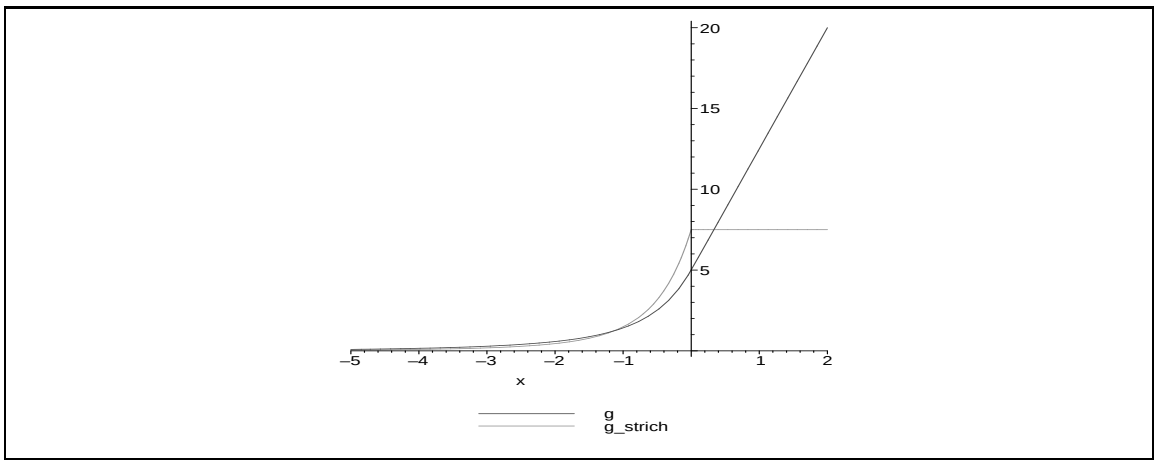


Abbildung 5.1: Funktion g und ihre Ableitung

5.3.1 Der Forward-Mode

In diesem Kapitel soll die zentrale Idee des sogenannten Forward-Mode der automatischen Differentiation erläutert werden. Wie im vorherigen Kapitel bereits gesagt, werden hier nur einfache Programme, also Funktionen der Art $P : \mathbb{R}^n \longrightarrow \mathbb{R}$ betrachtet, wobei die Funktion P aus einer Komposition von elementaren Funktionen $v_1, ..., v_m \in \Phi$ besteht. Als Beispiel dient hier die Funktion ([8])

$$f : \mathbb{R}^2 \longrightarrow \mathbb{R}$$

$$f(x_1, x_2) = (\sin(x_1/x_2) + x_1/x_2 - \exp(x_2)) \cdot (x_1/x_2 - \exp(x_2)). \tag{5.2}$$

Bei der numerischen Berechnung von $f(1.5, 0.5)$ wird die Auswertung wie in Abbildung 5.1 vorgenommen.

v_{-1}	$= x_1$	$= 1.5$	
v_0	$= x_2$	$= 0.5$	
v_1	$= v_{-1}/v_0$	$= 1.5/0.5$	$= 3.0$
v_2	$= \sin(v_1)$	$= \sin(3.0)$	$= 0.1411$
v_3	$= \exp(v_0)$	$= \exp(0.5)$	$= 1.6487$
v_4	$= v_1 - v_3$	$= 3 - 1.6487$	$= 1.3513$
v_5	$= v_2 + v_4$	$= 0.1411 + 1.3513$	$= 1.4924$
v_6	$= v_5 v_4$	$= 1.4924 \cdot 1.3513$	$= 2.0167$
y	$= v_6$	$= 2.0167$	

Tabelle 5.1: Auswertungsschema der Funktion f

Abbildung 5.2 zeigt den zugehörigen Berechnungsgraph.

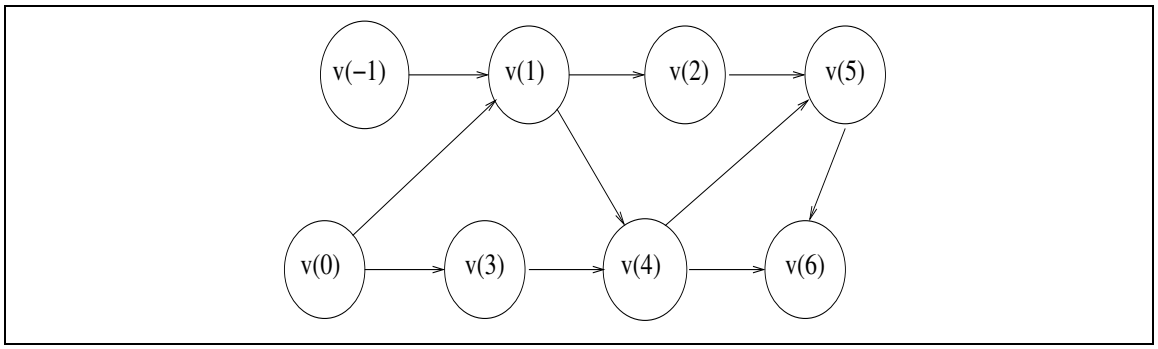


Abbildung 5.2: Berechnungsgraph zu Tabelle 5.1

Bei der Berechnung der partiellen Ableitung von f nach x_1 wird nach folgendem Schema vorgegangen:

Mit Hilfe der Kettenregel wird zunächst jede der "Hilfsvariablen" v_i partiell nach x_1 differenziert. Das Ergebnis wird mit $\dot{v}_i = \frac{\partial v_i}{\partial x_1}$ bezeichnet. Für die Eingabevariablen ergeben sich $\dot{v}_{-1} = 1$ und $\dot{v}_0 = 0$. Die weiteren partiellen Ableitungen können nun hieraus berechnet werden, z.B. ergibt sich für $\dot{v}_1$:

$$\dot{v}_1 = \frac{\partial v_1}{\partial v_{-1}} \frac{\partial v_{-1}}{\partial x_1} + \frac{\partial v_1}{\partial v_0} \frac{\partial v_0}{\partial x_1} = \frac{1}{v_0} \dot{v}_{-1} - \frac{v_{-1}}{v_0^2} \dot{v}_0 = 2$$

Werden die weiteren partiellen Ableitungen nach diesem Schema berechnet, ergeben sich die Ergebnisse aus Tabelle 5.2.

v_{-1}	$= x_1$	$= 1.5$	
$\dot{v}_{-1}$	$= \dot{x}_1$	$= 1$	
v_0	$= x_2$	$= 0.5$	
$\dot{v}_0$	$= \dot{x}_2$	$= 0$	
v_1	$= v_{-1}/v_0$	$= 1.5/0.5$	$= 3.0$
$\dot{v}_1$	$= (v_0\dot{v}_{-1} - v_{-1}\dot{v}_0)/v_0^2$	$= 1/0.5$	$= 2.0$
v_2	$= \sin(v_1)$	$= \sin(3.0)$	$= 0.1411$
$\dot{v}_2$	$= \cos(v_1)\dot{v}_1$	$= -0.99 \cdot 2$	$= -1.9800$
v_3	$= \exp(v_0)$	$= \exp(0.5)$	$= 1.6487$
$\dot{v}_3$	$= v_3\dot{v}_0$	$= 1.6487 \cdot 0$	$= 0$
v_4	$= v_1 - v_3$	$= 3 - 1.6487$	$= 1.3513$
$\dot{v}_4$	$= \dot{v}_1 - \dot{v}_3$	$= 2 - 0$	$= 2$
v_5	$= v_2 + v_4$	$= 0.1411 + 1.3513$	$= 1.4924$
$\dot{v}_5$	$= \dot{v}_2 + \dot{v}_4$	$= -1.9800 + 2$	$= 0.0200$
v_6	$= v_5v_4$	$= 1.4924 \cdot 1.3513$	$= 2.0167$
$\dot{v}_6$	$= \dot{v}_5v_4 + v_5\dot{v}_4$	$= 0.02 \cdot 1.3513 + 1.4924 \cdot 2$	$= 3.0118$
y	$= v_6$	$= 2.0167$	
$\dot{y}$	$= \dot{v}_6$	$= 3.0118$	

Tabelle 5.2: Vorwärts-Ableitung der Funktion f

Eine ausführlichere Beschreibung des Forward-Mode findet sich in [8].

5.3.2 Der Reverse-Mode

Ein Kennzeichen des Forward-Mode ist, dass eine Eingangsvariable x_i gewählt wird und die partielle Ableitung des Programms nach dieser Variable berechnet wird. Der Reverse-Mode geht den umgekehrten Weg: Hier wird eine Ausgangsvariable y_i gewählt und es werden die partiellen Ableitungen bezüglich dieser Variable berechnet. Die "Zwischenvariablen" v_i werden ersetzt durch $\bar{v}_i$ mit $\bar{v}_i = \frac{\partial y}{\partial v_i}$.

Ein Vorteil des Reverse-Mode ist somit, dass stets die Ableitungen bezüglich aller Eingangsvariablen bestimmt werden. In Maple sind die Algorithmen für Forward-Mode und Reverse-Mode in der Funktion `GRADIENT` im Paket `codegen` implementiert. Der im vorangegangenen Kapitel vorgestellte Operator `D` kann nur einfache Programme im Forward-Mode verarbeiten. Durch den Reverse-Mode werden in vielen Fällen effizientere Programme erzeugt. Ein Beispiel soll dies verdeutlichen. Das Programm sei gegeben durch:

```
f:=proc(x,y)
  local u_1, u_2, u_3;
  u_1 := x^2*y;
  u_2 := y^2*x^2;
  u_3 := x*y^2;
  4*u_1+u_2*u_3+8*u_3^2;
end proc:
```

Der Gradient dieses Programms soll mit automatischer Differentiation berechnet werden. Die Berechnung erfolgt zuerst mit dem Reverse-Mode:

```
> fr := GRADIENT(f);
```

$fr := \mathbf{proc}(x, y)$
local $dfr0, u\_1, u\_2, u\_3$;
$u\_1 := x^2 * y$;
$u\_2 := y^2 * x^2$;
$u\_3 := x * y^2$;
$dfr0 := \mathrm{array}(1..3)$;
$dfr0_3 := u\_2 + 16 * u\_3$;
$dfr0_2 := u\_3$;
$dfr0_1 := 4$;
return $dfr0_3 * y^2 + 2 * dfr0_2 * x * y^2 + 2 * dfr0_1 * x * y,$
$2 * dfr0_3 * x * y + 2 * dfr0_2 * x^2 * y + dfr0_1 * x^2$
end proc

Zum Vergleich erfolgt nun die Berechnung im Forward-Mode:

```
> ff := GRADIENT(f, mode=forward);
```

$ff := \mathbf{proc}(x, y)$
local $du\_1, du\_2, du\_3, u\_1, u\_2, u\_3$;
$du\_1 := \mathrm{array}(1..2)$;
$du\_2 := \mathrm{array}(1..2)$;
$du\_3 := \mathrm{array}(1..2)$;
$du\_1_1 := 2 * x * y$;
$du\_1_2 := x^2$;
$u\_1 := x^2 * y$;
$du\_2_1 := 2 * x * y^2$;
$du\_2_2 := 2 * x^2 * y$;
$u\_2 := y^2 * x^2$;
$du\_3_1 := y^2$;
$du\_3_2 := 2 * x * y$;
$u\_3 := x * y^2$;
return $4 * du\_1_1 + u\_3 * du\_2_1 + (u\_2 + 16 * u\_3) * du\_3_1,$
$4 * du\_1_2 + u\_3 * du\_2_2 + (u\_2 + 16 * u\_3) * du\_3_2$
end proc

Um zu ermitteln, wie kostenintensiv die einzelnen Funktionsaufrufe sind, kann die Funktion `cost` verwendet werden. Sie gibt unter anderem an, wie viele arithmetische Operationen, lokale Variablen, etc. verwendet wurden.

```
> cost(fr);
```

$6\ storage + 7\ assignments + 26\ multiplications + 9\ subscripts + 5\ additions$

```
> cost(ff);
```

$9\ storage + 12\ assignments + 27\ multiplications + 12\ subscripts + 6\ additions$

Weitere Informationen zur automatischen Differentiation, insbesondere für Maple, finden sich in [11].

5.4 Effizienzvergleich

Im Folgenden wird ein Vergleich zwischen automatischer und symbolischer Differentiation im Rahmen von Maple durchgeführt. Der Vergleich bezieht sich dabei auf die benötigte CPU-Zeit und den verwendeten Speicherplatz. Zunächst geht es um die Ableitung der folgenden Funktion:

```
> g := proc(x,n::nonnegint)
    local i, t;
    t := x;
    for i to n do
      t := x^tan(t);
    end do:
    t
end proc:
```

Für festes $n \in \mathbb{N}$ ergeben sich damit Funktionen der Art (hier beispielhaft für $n = 5$):

$$x^{\left(\tan\left(x^{\left(\tan\left(x^{\tan\left(x^{\tan(x)}\right)}\right)\right)}\right)\right)}$$

Mit Hilfe symbolischer und automatischer Differentiation soll nun die dritte Ableitung von g an der Stelle $x = 0.9$ für $n = 15$ berechnet werden. Dabei wird einerseits die benötigte CPU-Zeit und andererseits der verwendete Hauptspeicher festgehalten [1].
Mit Hilfe automatischer Differentiation ergibt sich:

```
> setbytes:=kernelopts(bytesused):
  settime:=time():
  D[1$3](g)(0.9, 15);
  CPU_time1 := ( time()-settime)*Sekunden;
  memory_used1:=evalf((kernelopts(bytesused)-setbytes)/10^6,6)*MB;
```

$$28.50114134$$
$$CPU\_time1 := 0.100\, Sekunden$$
$$memory\_used1 := 1.72240\, MB$$

Mit symbolischer Differentiation ergibt sich:

```
> setbytes:=kernelopts(bytesused):
  settime:=time():
  eval( diff(g(x,15),x$3), x=0.9);
  CPU_time2 := (time()-settime)*Sekunden;
  memory_used2:=evalf((kernelopts(bytesused)-setbytes)/10^6,6)*MB;
```

$$28.50114134$$
$$CPU\_time2 := 11.430\, Sekunden$$
$$memory\_used2 := 59.5523\, MB$$

Um die Ergebnisse zu verdeutlichen, werden die Quotienten

$$Quotient\_CPU = \frac{CPU\_time2}{CPU\_time1} = \frac{11.430s}{0.100s} = 114.3$$

und

$$Quotient\_MEM = \frac{memory\_used2}{memory\_used1} = \frac{59.5523MB}{1.72240MB} \approx 34.58$$

[1] Als Testsystem diente ein unter SuSE Linux 8.0 laufender AMD-Athlon PC mit 1GHz und 256 MB Hauptspeicher

gebildet. Hier ist deutlich zu erkennen, dass die automatische Differentiation der symbolischen Differentiation sowohl im Hinblick auf die benötigte CPU-Zeit als auch im Hinblick auf den verwendeten Hauptspeicherbedarf deutlich überlegen ist.

Die Funktionen, die im Zusammenhang mit dem Lösen von Ausgleichsproblemen relevant sind, haben eine deutlich einfachere Gestalt. Betrachten wir hier nochmals die Funktion (5.2):

```
> h := proc(x1,x2)
  local t1,t2,t3,t4,t5,t;
    t1 := x1/x2;
    t2 := sin(t1);
    t3 := exp(x2);
    t4 := t1-t3;
    t5 := t2+t4;
    t  := t5*t4;
  end proc:
```

Hier wurden die gleichen Messungen vorgenommen und die Resultate festgehalten. Um vernünftige Messergebnisse zu erhalten, wird das eigentliche Differenzieren in einer Schleife durchgeführt. Zunächst die automatische Differentiation:

```
> setbytes:=kernelopts(bytesused):
  settime:=time():
  for i from 1 to 100 do
    D[1](h)(1.5, 0.5);
  end do:
  CPU_time1 := ( time()-settime)*Sekunden;
  memory_used1 := evalf(( kernelopts(bytesused)-setbytes)/10^6,6)*MB;
```

$$CPU\_time1 := 0.959\, Sekunden$$
$$memory\_used1 := 10.5090\, MB$$

Die symbolische Differentiation liefert:

```
> setbytes:=kernelopts(bytesused):
  settime:=time():
  for i from 1 to 100 do
    evalf( subs( x=1.5, y=0.5, diff(h(x,y), x) ) );
  end do:
  CPU_time2 := (time()-settime)*Sekunden;
  memory_used2 := evalf(( kernelopts(bytesused)-setbytes)/10^6,6)*MB;
```

$$CPU\_time2 := 0.020\, Sekunden$$
$$memory\_used2 := 0.2853\, MB$$

Auch hier werden die obrigen Quotienten gebildet:

$$Quotient\_CPU = \frac{CPU\_time2}{CPU\_time1} = \frac{0.020s}{0.959s} \approx 0.021$$

und

$$Quotient\_MEM = \frac{memory\_used2}{memory\_used1} = \frac{0.2853MB}{10.5090MB} \approx 0.027$$

Die Ergebnisse zeigen in diesem Fall Vorteile für die symbolische Differentiation. Da die Funktionen bei der Berechnung nichtlinearer Ausgleichsprobleme von der Gestalt des zweiten Beispiels sind, waren diese Ergebnisse mit ein Grund, sich im entwickelten Programmpaket für die Methode der symbolischen Differentiation zu entscheiden.

Kapitel 6

Funktionalität des Programmpakets

6.1 Überblick

In das zu dieser Diplomarbeit entstandene Programm fließen die Elemente aus den bisherigen Kapiteln ein. Einen Überblick über die kombinierten Elemente gibt Abbildung 6.1.

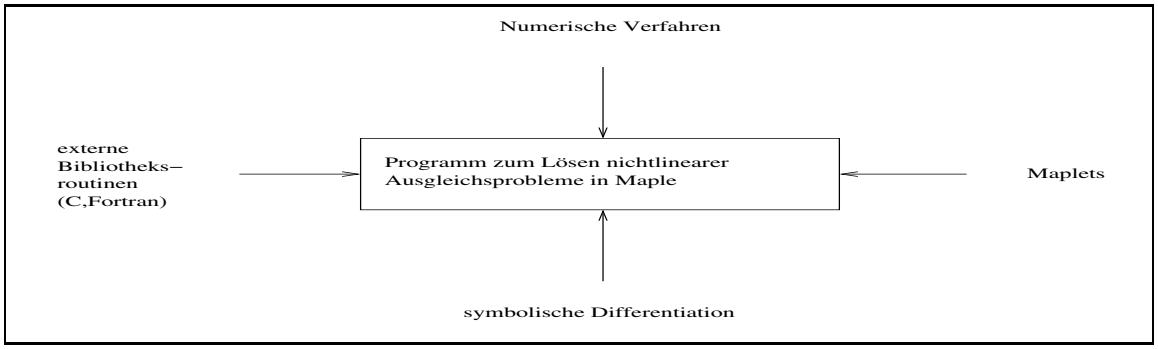


Abbildung 6.1: Im Programmpaket verwendete Elemente

Für die eigentliche numerische Lösung der nichtlinearen Ausgleichsprobleme werden Bibliotheksfunktionen benutzt. Zwei Verfahren sind implementiert: Der modifizierte Gauß-Newton-Algorithmus stammt aus der NAG Fortran90 Library ([18]). Die Implementierung des Levenberg-Marquardt-Algorithmus wurde aus den Numerical Recipes ([5]) entnommen. Beide Algorithmen können wie in Kapitel 4 beschrieben in das Maple-System integriert werden.
In Kapitel 5 wurden zwei Methoden vorgestellt, mit denen Ableitungen (bzw. partielle Ableitungen) von Funktionen berechnet werden können. Hier standen sich die symbolische Differentiation und die automatischen Differentiation gegenüber. Bei der Implementierung des Programms wurde die Methode der symbolischen Differentiation gewählt. Grund für diese Entscheidung war, dass die Verwendung der automatischen Differentiation einen größeren Implementierungsaufwand erfordert hätte. Zudem ergeben sich für die hier verwendeten Modellfunktionen bei der Differentiation leichte Vorteile für die symbolische Differentiation, wie in 5.4 gezeigt wurde.

Das Programmpaket kann grob in zwei Teilbereiche gegliedert werden. Zum einen in die numerischen Verfahren und zum anderen in die darüberliegende graphische Oberfläche. Die numerischen Verfahren sind dabei unabhängig von der graphischen Oberfläche, können also auch in der Maple-Worksheet-Umgebung verwendet werden.

Das Zusammenspiel der Elemente beim Aufruf der numerischen Algorithmen von Maple aus soll im Folgenden beschrieben werden. Beim Levenberg-Marquardt-Algorithmus wird wie in Abbil-

dung 6.2 verfahren. Hier existiert bereits auf der externen Ebene ein Hauptprogramm. Der Hauptalgorithmus befindet sich in der Datei *Hauptalgorithmus*. Die übergebene Modellfunktion wird nun symbolisch nach den Parametern $a_1, ..., a_M$ differenziert, und die Ergebnisse in das File *Ableitungen* geschrieben. Mit Hilfe des `system` Kommandos wird nun die Library erzeugt und diese mit Hilfe des `define_external` Kommandos in das Maple-System eingebunden, woraus sie nun aufgerufen werden kann.

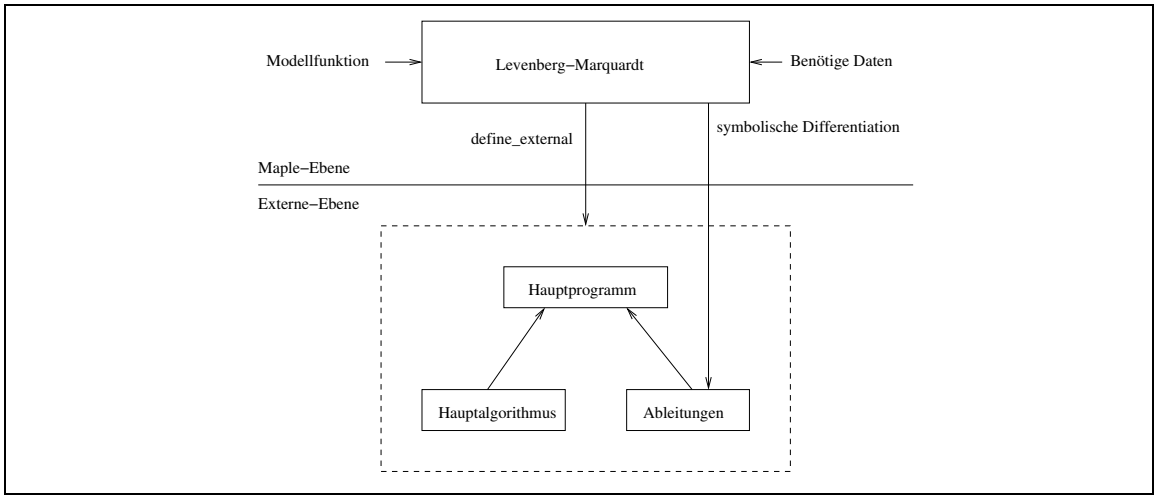


Abbildung 6.2: Prinzipieller Ablauf des Levenberg-Marquardt-Algorithmus

Bei dem modifizierten Gauß-Newton-Algorithmus wird ähnlich vorgegangen (Abbildung 6.3). Hier werden die symbolischen Ableitungen nicht separat gespeichert, sondern werden mit dem Code des Hauptprogramms direkt in das File *Hauptprogramm* geschrieben. Nun kann wieder mit Hilfe des `system` Kommandos die Library unter Einbeziehung der NAG-Fortran90-Library erzeugt werden.

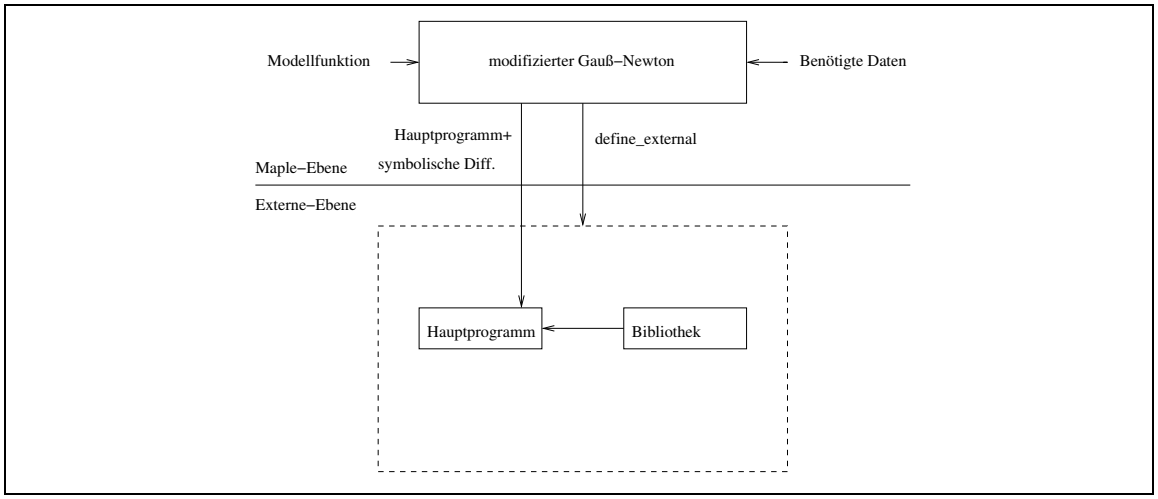


Abbildung 6.3: Prinzipieller Ablauf des modifizierten Gauß-Newton-Algorithmus

Ab der Version Maple 8 ist das Programmpaket in vollem Umfang benutzbar. In vorherigen Versionen ist die Benutzung eingeschränkt. Release 7 verfügt nur durch ein Zusatz-Paket über ein GUI-Tool. In früheren Versionen fehlt es völlig.

Im Folgenden wird zunächst die Worksheet-orientierte Benutzung beschrieben. Im Anschluß daran erfolgt die Beschreibung des Programms unter Benutzung von Maplets.
Das komplette Programmpaket ist über mehrere Module verteilt worden. Einen Überblick über die Hauptmodule gibt Abbildung 6.4.

Die Hauptfunktionalität der graphischen Oberfläche ist im Modul `NonlinearFit` implementiert. Hilfsroutinen sind gesondert im Modul `NonlinearFitTools` gespeichert. Zur Einbindung der Algorithmen zur nichtlinearen Ausgleichsrechnung wird das Modul `NonlinearFit_Linux`

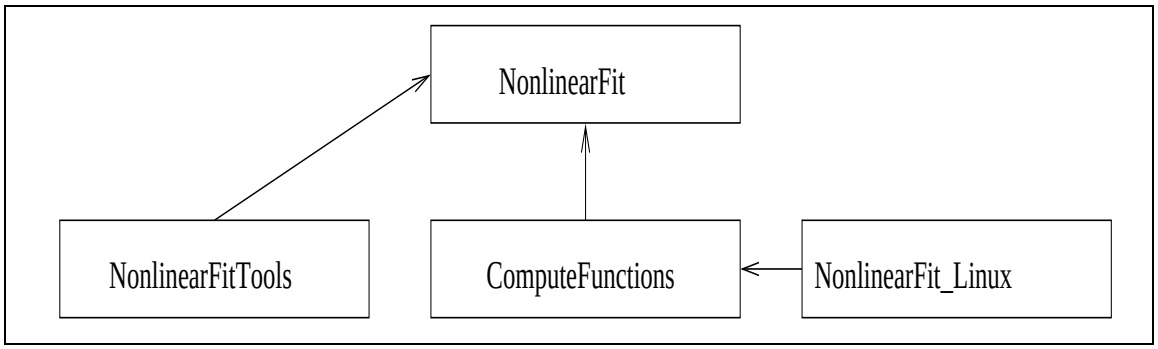


Abbildung 6.4: Module und deren Abhängigkeiten

benötigt. Wie der Name schon sagt, ist dieses Modul für die Verwendung unter dem Betriebssystem Linux konzipiert. Über das Zusatzmodul `ComputeFunctions` können diese Algorithmen von der graphischen Benutzeroberfläche aus verwendet werden.

6.2 Verwendung im Worksheet

Die Routinen für die nichtlineare Ausgleichsrechnung können sowohl im Rahmen der graphischen Oberfläche als auch in der normalen Maple Worksheet-Umgebung genutzt werden. Um die Routinen direkt aus dem Maple-System benutzen zu können, muss das Paket `NonlinearFit_Linux` mit eingebunden werden. Jetzt stehen zwei neue Routinen zur Verfügung:
`LevenbergMarquardt` für den Levenberg-Marquardt-Algorithmus (siehe 2.3.3), und
`ModifiedGaussNewton` für einen modifizierten Gauß-Newton-Algorithmus (siehe 2.3.2). Im Folgenden wird kurz darauf eingegangen, wie die Algorithmen von Maple aus aufzurufen sind.

ModifiedGaussNewton(f,x,y,a,max_iter,path_to_source,res,a_fitted,used_iter, cov_matrix)
Levenberg_Marquardt(f,x,y,sig,a,fix,max_iter,path_to_source,res,a_fitted,used_iter, cov_matrix)

Tabelle 6.1: Aufruf der Algorithmen innerhalb eines Maple-Worksheets

Die Tabelle 6.1 zeigt die Funktionsdefinitionen der beiden Prozeduren. Eine Beschreibung der Parameter findet sich in Tabelle 6.2.

Parameter	Bedeutung
f	Modellfunktion $f(\mathbf{x}_1, ..., \mathbf{x}_N, a_1, ..., a_M)$
x	Liste der (mehrdimensionalen) X-Werte
y	Liste der (eindimensionalen) Y-Werte
sig	Individuelle Gewichtung der Punkte $\mathbf{x}_1, ..., \mathbf{x}_N$
a	Liste mit den Startwerten für die Parameter $a_1, ..., a_M$
fix	Liste, wobei eine 0 an Position i bedeutet, dass der Parameter a_i festgehalten wird
max_iter	Anzahl der Iterationen, die maximal ausgeführt werden soll
path_to_source	Pfadangabe zu den C/Fortran Quellcodes
a_fitted	Vom Iterationsverfahren berechnete Parameter
used_iter	Ausgeführte Iterationen bis zum Erreichen des Abbruchkriteriums
cov_matrix	Vom Iterationsverfahren genäherte Kovarianzmatrix

Tabelle 6.2: Beschreibung der Parameter

Im Folgenden soll der Aufruf beider Algorithmen an jeweils einem Beispiel demonstriert werden. Beide hier verwendeten Datensätze sind im Anhang A dokumentiert. Die erste Beispielsitzung bezieht sich auf den Levenberg-Marquardt-Algorithmus und den Datensatz A.2:

```
> with( NonlinearFit_Linux);
```

$NonlinearFit\_Linux :=$ **module**()
local $LevenbergMarquardt\_module$, $ModifiedGaussNewton\_module$;
export $LevenbergMarquardt$, $ModifiedGaussNewton$;
description "Solving nonlinear fitting problems";

end module

$[LevenbergMarquardt, ModifiedGaussNewton]$

```
> # Modellfunktion
> g := a[1] / ( 1 + a[2] * exp(-a[3]*x[1]));
```

$$g := \frac{a_1}{1 + a_2\, e^{(-a_3\, x_1)}}$$

```
> # X-Werte
> xvalues := [seq([i], i=1..12)];
```

$xvalues := [[1], [2], [3], [4], [5], [6], [7], [8], [9], [10], [11], [12]]$

```
> # Y-Werte
> yvalues := [494, 759, 1016, 1215, 1619, 2964, 8489, 22377,
> 32618, 38153, 41462, 42375];
```

$$yvalues :=$$
$$[494, 759, 1016, 1215, 1619, 2964, 8489, 22377, 32618, 38153, 41462, 42375]$$

```
> # Gewichte
> weights := [seq((1/j)^ 2, j=1..12)];
```

$$weights := [1, \frac{1}{4}, \frac{1}{9}, \frac{1}{16}, \frac{1}{25}, \frac{1}{36}, \frac{1}{49}, \frac{1}{64}, \frac{1}{81}, \frac{1}{100}, \frac{1}{121}, \frac{1}{144}]$$

```
> # Startparameter
> guess := [1000,500,1];
```

$$guess := [1000, 500, 1]$$

```
> # Alle Parameter
> fixed := [1,1,1];
```

$$fixed := [1, 1, 1]$$

```
> # Verzeichnis des Sourcecodes
> PathToSource := "/home/kuelheim/msw2002/sources";
```

$$PathToSource := \text{“/home/kuelheim/msw2002/sources”}$$

```
> # Number of iterations
> Max_Iterations := 40;
```

$$Max\_Iterations := 40$$

```
> f := unapply( NonlinearFit_Linux:-LevenbergMarquardt( g,
> xvalues, yvalues, weights , guess,  fixed,   Max_Iterations,
> PathToSource, 'outRes', 'FittedParameters', 'Used_iter',
> 'CoVar'),x[1]);
```

“Building function...”

“Compiling functions...”

“Calling C function...”

$$[42592.2865666323196, 12046.7970306885618, 1.17431983616175727]$$

$$f := x\_1 \rightarrow \frac{0.177661719482637978\,10^{11}}{42592.2865666323196 + 12046.7970306885618\,e^{(-1.17431983616175727\,x\_1)}}$$

```
> outRes;
> Used_iter;
```

$$0.177661719482637978\,10^{11}$$
$$26$$

Abbildung 6.5 zeigt die Datenpunkte und die Modellfunktion für die angepaßten Parameter.

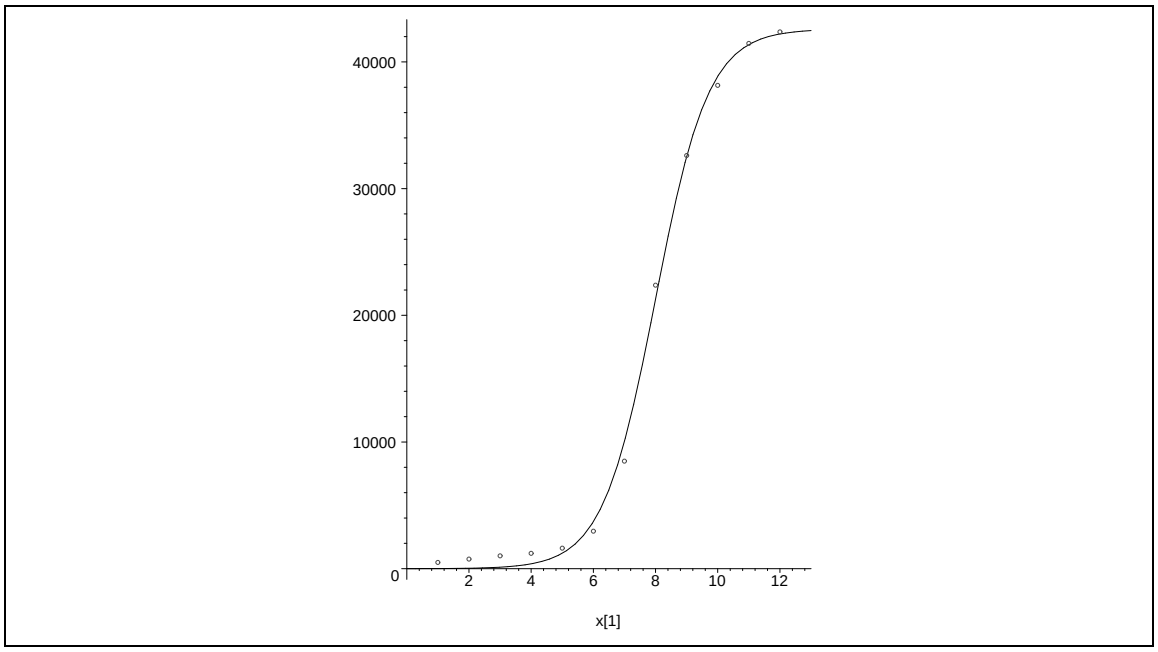


Abbildung 6.5: Graphische Darstellung der Datenpunkte und der Modellfunktion $g := \frac{a_1}{1+a_2 \exp -a_3 x_1}$ für die mit dem Levenberg-Marquardt-Algorithmus optimierten Parametern

Die nächste Beispielsitzung bezieht sich auf den modifizierten Gauß-Newton-Algorithmus und Datensatz A.3:

```
> with( NonlinearFit_Linux);
```

$NonlinearFit\_Linux :=$ **module**()
local $LevenbergMarquardt\_module$, $ModifiedGaussNewton\_module$;
export $LevenbergMarquardt$, $ModifiedGaussNewton$;
description "Solving nonlinear fitting problems";

end module

$[LevenbergMarquardt,\ ModifiedGaussNewton]$

```
> # Modellfunktion
> f := a[1] + x[1]/(a[2]*x[2] + a[3]*x[3]);
```

$$f := a_1 + \frac{x_1}{a_2\, x_2 + a_3\, x_3}$$

```
> # X-Werte
> xvalues := [
> [1,15,1],[2,14,2],[3,13,3],[4,12,4],[5,11,5],[6,10,6],[7,9,7],
> [8,8,8],[9,7,7],[10,6,6],[11,5,5],[12,4,4],[13,3,3], [14,2,2],
> [15,1,1] ];
```

$xvalues := [[1, 15, 1], [2, 14, 2], [3, 13, 3], [4, 12, 4], [5, 11, 5], [6, 10, 6], [7, 9, 7], [8, 8, 8], [9, 7, 7], [10, 6, 6], [11, 5, 5], [12, 4, 4], [13, 3, 3], [14, 2, 2], [15, 1, 1]]$

```
> # Y-Werte
> yvalues := [ 0.14, 0.18, 0.22, 0.25, 0.29, 0.32, 0.35, 0.39,
> 0.37, 0.58, 0.73, 0.96, 1.34, 2.10, 4.39];
```

$yvalues := [0.14, 0.18, 0.22, 0.25, 0.29, 0.32, 0.35, 0.39, 0.37, 0.58, 0.73, 0.96, 1.34, 2.10, 4.39]$

```
> # Startparameter
> guess := [0.5, 1.0, 1.5];
```

$$guess := [0.5, 1.0, 1.5]$$

```
> # Verzeichnis des Sourcecodes
> PathToSource := "/home/kuelheim/msw2002/sources";
```

$$PathToSource := \text{“/home/kuelheim/msw2002/sources”}$$

```
> # Maximale Iterationen
> Max_Iter := 40;
```

$$Max\_Iter := 40$$

```
> func := ModifiedGaussNewton( f, xvalues, yvalues, guess,
> Max_Iter, PathToSource, 'outRes', 'outFittedParams',
> 'Used_Iterations', 'Cov_matrix');
```

“BuildFunction...”

“CompileFunction...”

“Result von NAG: ”, 0

$$func := 0.0824105598097717162 + \frac{x_1}{1.13303609205156009\,x_2 + 2.34369517871324451\,x_3}$$

```
> outRes;
> outFittedParams;
> Used_Iterations;
```

$$0.00821487730657897985$$

$$[0.0824105598097717162,\ 1.13303609205156009,\ 2.34369517871324451]$$

$$6.$$

Da die Funktion selbst nicht dargestellt werden kann, zeigt Abbildung 6.6 eine Darstellung der Residuen in Abhängigkeit vom Index des Messpunktes, d.h. bei gegebenem $\mathbf{x} \in \mathbb{R}^3$ aus dem Datensatz die Abweichung zwischen angegebenem y-Wert und dem Funktionswert der Modellfunktion. Die erkennbare Systematik lässt darauf schließen, dass die Modellfunktion noch verbessert werden könnte. Zudem scheint der neunte Punkt mit hoher Wahrscheinlichkeit ein Ausreißer zu sein.

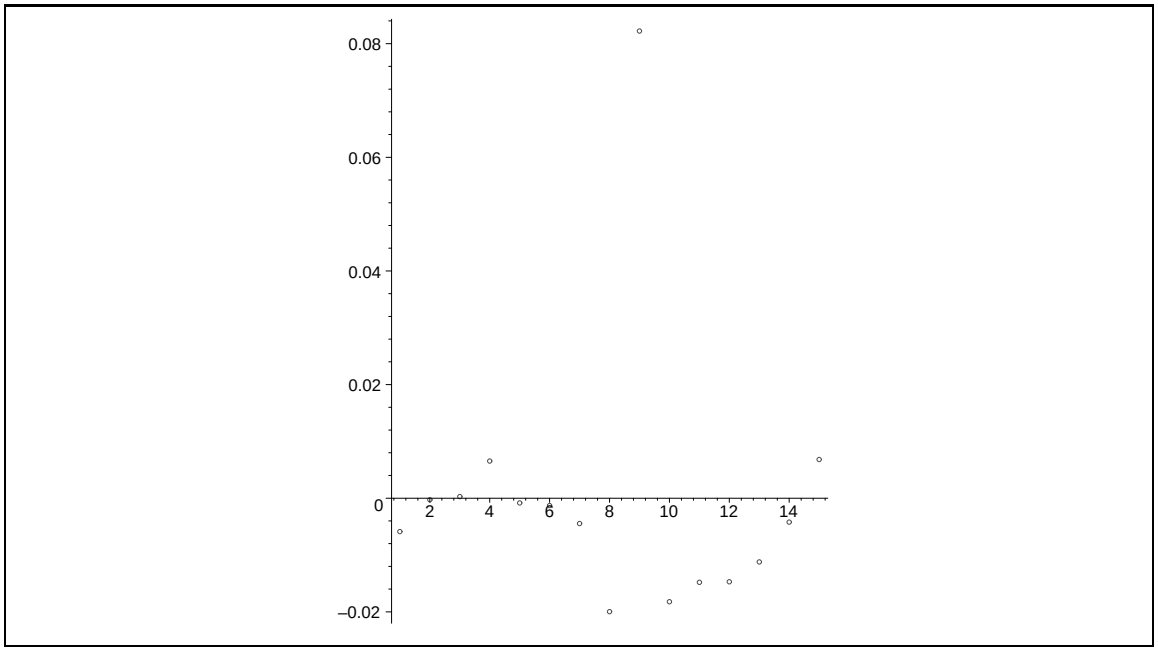


Abbildung 6.6: Residuen nach Anpassung mit dem modifizierten Gauß-Newton-Algorithmus für die Modellfunktion $f := a_1 + \frac{x_1}{a_2 x_2 + a_3 x_3}$

6.3 Die graphische Oberfläche

Das Hauptprogramm für die graphische Oberfläche befindet sich in dem Modul `NonlinearFit`. In diesem Programmpaket befindet sich die Funktion `NonlinearFitMaplet`, mit der die Oberfläche gestartet wird. Ein Bild dieser Oberfläche sieht man in Abbildung 6.7.

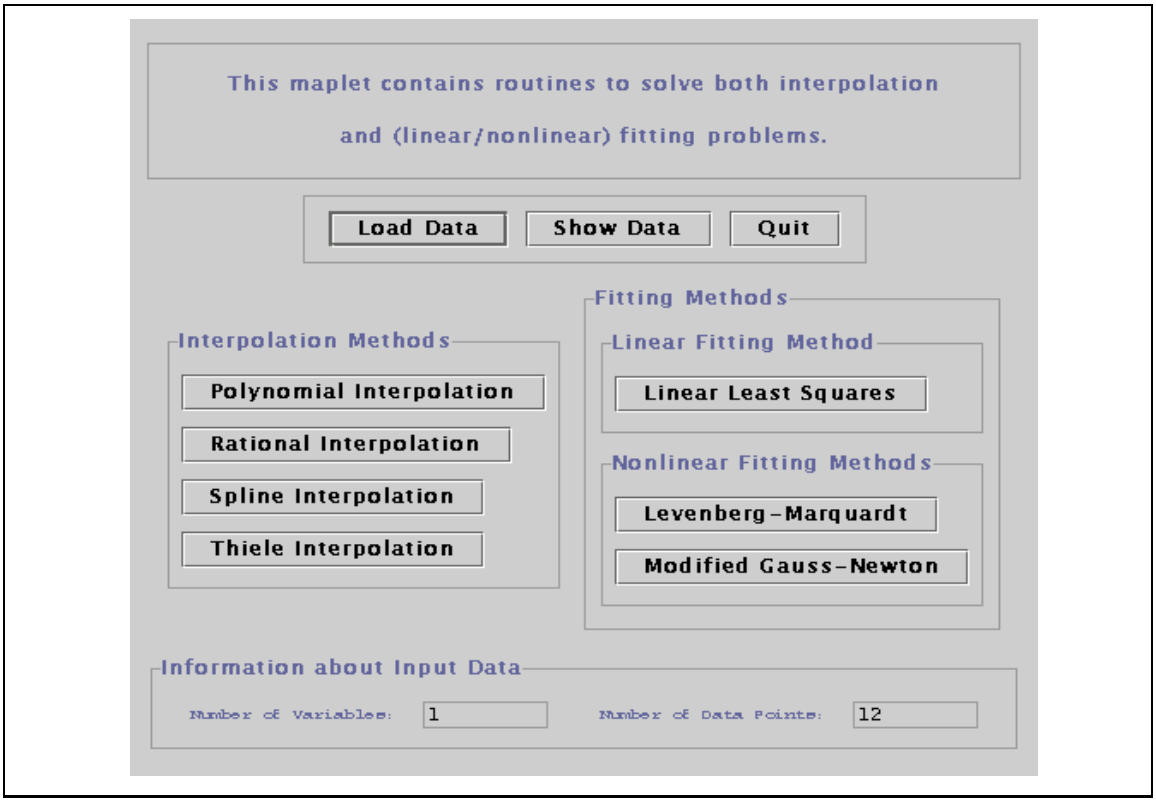


Abbildung 6.7: Eingangspanel des Programmpaketes

In der obersten Zeile sind Grundoperationen wie das Laden und Anzeigen von Daten bzw. das Beenden der Anwendung implementiert. Mathematische Verfahren, mit dem die Daten bearbeitet werden können, befinden sich in der Mitte der Abbildung. Auf der linken Seite sind die Interpolationsmethoden zu finden. Diese beinhalten:

1. Polynomiale Interpolation
2. Rationale Interpolation
3. Interpolation mit Splines
4. Thiele Interpolation

Auf der rechten Seite finden sich Verfahren zum Lösen von Ausgleichsproblemen. Dazu zählt:

1. Lineare Ausgleichsrechnung
2. Nichtlineare Ausgleichsrechnung
   (a) Levenberg-Marquardt-Verfahren
   (b) Modifiziertes Gauß-Newton-Verfahren

In der letzten Zeile befinden sich Informationen über den eingelesenen Datensatz. Angegeben werden hier die Anzahl der Variablen und die Anzahl der eingelesenen Datenpunkte.
Um mit den angegebenen Verfahren arbeiten zu können, muss zunächst ein Datensatz eingelesen werden. Der generelle Aufbau eines Datensatzes wird in Anhang B beschrieben. Die Verfahren für nichtlineare Ausgleichsprobleme werden in einem der nächsten Abschnitte noch genauer beschrieben. Die Interpolationsverfahren stellen zwar nicht das Hauptthema dieser Arbeit dar, sollen aber aus Gründen der Vollständigkeit hier am Beispiel der Spline-Interpolation vorgestellt werden. Alle

Interpolationsverfahren können nur im Fall einer unabhängigen Variablen benutzt werden. Abbildung 6.8 zeigt das graphische Ergebnis einer Interpolation. Die Oberflächen der übrigen Verfahren sehen ähnlich aus, haben jedoch je nach Art des Verfahrens und der benötigten Parameter noch andere Eingabemöglichkeiten.

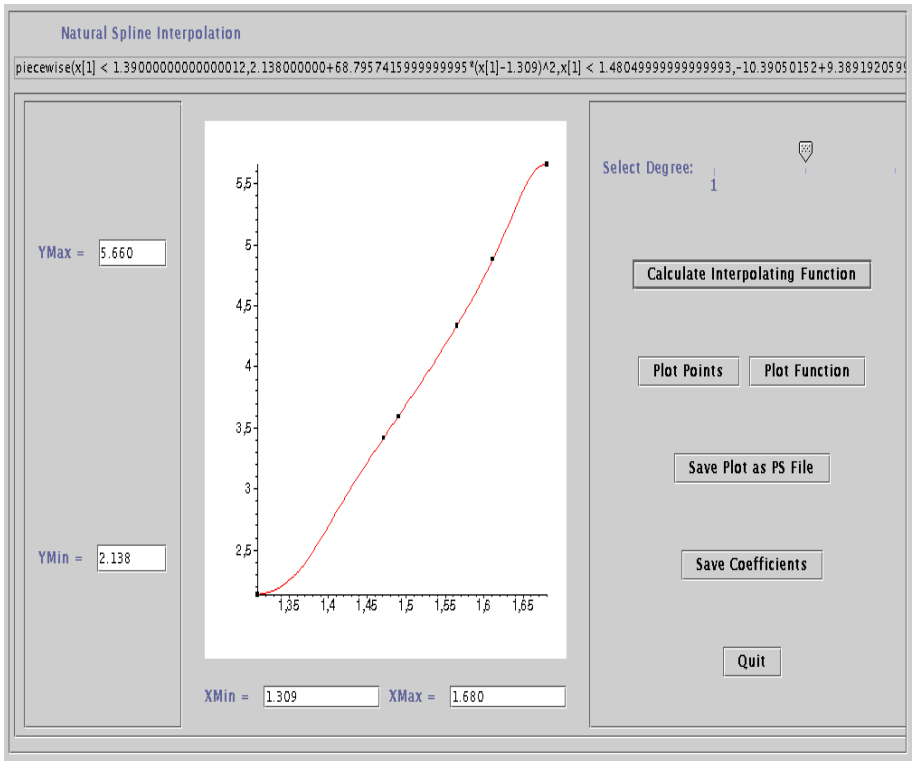


Abbildung 6.8: Spline Interpolation

Die Oberflächen der Verfahren haben allesamt die Möglichkeit, nach erfolgten Berechnungen die Punkte mit zugehöriger Funktion zu plotten. Die Plotbereiche können numerisch vorgegeben werden. Das Bild der Funktion kann als Postscript File gespeichert werden. Berechnete Resultate können in eine Textdatei geschrieben werden. Die Ergebnisse der Spline-Interpolation (die Koeffizienten der Spline-Polynome) sind in Abbildung 6.9 dargestellt, der verwendete Datensatz findet sich in Tabelle 6.3.

Nr.	x	y	ω
1	1.309	2.138	1
2	1.471	3.421	1
3	1.490	3.597	1
4	1.565	4.340	1
5	1.611	4.882	1
6	1.680	5.660	1

Tabelle 6.3: Datensatz DanWood [7]

```
x[1] < 1.390000000
120.018196
-180.107252
68.795742

x[1] < 1.480500000
-33.841704
41.273899
-10.837766

x[1] < 1.527500000
3.020364
-8.522882
5.979790

x[1] < 1.588000000
23.091587
-34.802717
14.582027

x[1] < 1.645500000
82.913660
-110.145378
38.304527

1.645500000 <= x[1]
-645.325355
774.982566
-230.649573
```

Abbildung 6.9: Berechnete Koeffizienten des interpolierenden kubischen Splines

6.4 Nichtlineare Ausgleichsrechnung

6.4.1 Ergebnisse für den eindimensionalen Fall

Die Maplet-Funktionalität des Programms soll zunächst an einem eindimensionalen Ausgleichsproblem demonstriert werden. Als Datensatz dient hier nochmals der in Tabelle 6.3 angegebene Datensatz. Die Modellfunktion hat die Gestalt:

$$f(x) = a[1] \cdot x[1]^{a[2]}$$

Die Ergebnisse der Rechnung sowie eine graphische Ausgabe der Funktion mit den optimalen Parametern mit dem Levenberg-Marquardt-Algorithmus zeigt Abbildung 6.10.

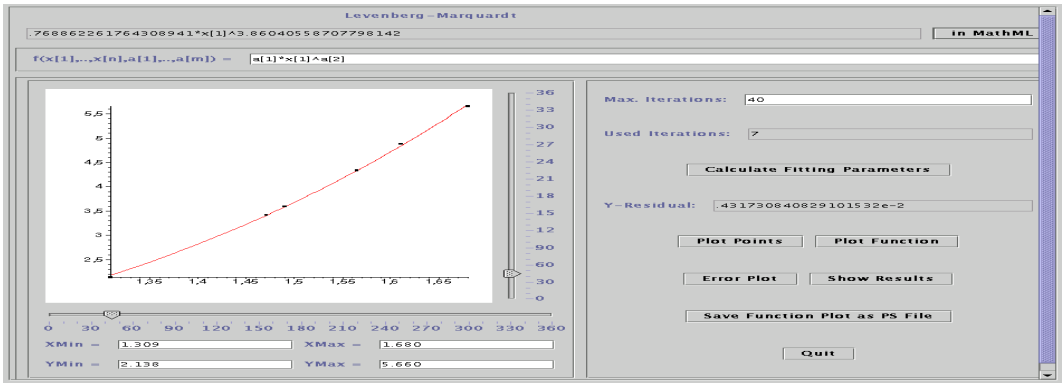


Abbildung 6.10: Graphische Ausgabe zum eindimensionalen Ausgleichsproblem

Außer den graphischen Resultaten stehen nach ausgeführter Ausgleichsrechnung als numerische Resultate 95% Konfidenzintervalle für die bestimmten Parameter und die geschätzte Kovarianzmatrix zur Verfügung. Diese sind in Abbildung 6.11 dargestellt. Berechnungsvorschriften finden sich in [21]. Die geschätzte Kovarianzmatrix gibt Auskunft über die verwendete Modellfunktion. Die Diagonale dieser Matrix stellt die (geschätzten) Varianzen der Parameter $a_i,\ 1 \leq i \leq M$ dar. Wenn C die Kovarianzmatrix bezeichnet, so geben die Nebendiagonalelemente $C_{ij},\ 1 \leq i,j \leq M, i \neq j$ die Kovarianz zwischen a_i und a_j wieder. Aus der Kovarianz kann der Pearsonsche Korellationskoeffizient der beiden Parameter berechnet werden, mit dem Aussagen über den linearen Zusammenhang der beiden Parameter getroffen werden können.

```
Starting values:              Fitted values:
1.000000                      a[1] = .768862
1.000000                      a[2] = 3.860406

Y-Residual: 4.317308e-03

95 % Confidence intervals:
[            .646 ;             .891]
[          3.514 ;            4.207]

Covariance-Matrix:
            .31              -.868
           -.868             2.479
```

Abbildung 6.11: Numerische Ergebnisse für das eindimensionale Ausgleichsproblem

6.4.2 Ergebnisse für den zweidimensionalen Fall

Als zweites Beispiel soll eine Ausgleichsrechnung für den mehrdimensionalen Fall durchgeführt werden. Als Datensatz dient hier der in Tabelle A.1 angegebene.

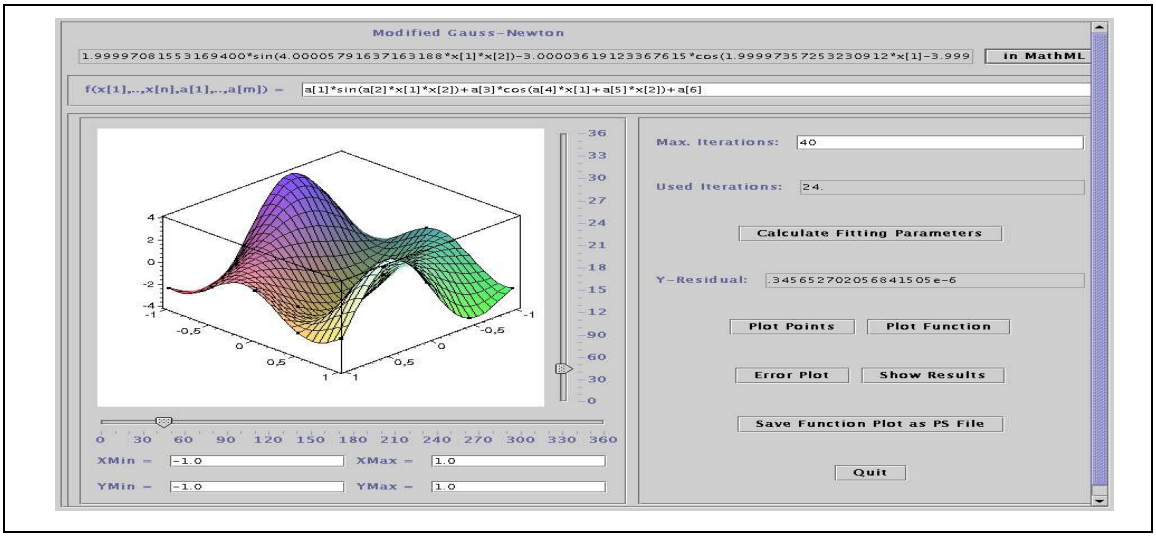


Abbildung 6.12: Graphische Ausgabe zum zweidimensionalen Ausgleichsproblem

Die Modellfunktion lautet hier:

$$f(x[1], x[2]) = a[1]\sin(a[2]x[1]x[2]) + a[3]\cos(a[4]x[1] + a[5]x[2]) + a[6]$$

Abbildung 6.12 zeigt das Resultat der Ausgleichsrechnung für die Startparameter $a[1] = a[2] = a[3] = a[4] = a[5] = a[6] = 1$.

Die Ausgleichsrechnung ergibt als Werte für die Parameter:

```
Fitted values:
a[1] = -1.999971
a[2] = -4.000058
a[3] = -3.000036
a[4] = -1.999974
a[5] = 4.000000
a[6] = -.999986
```

Vergleicht man diese Werte mit den Werten in Abbildung 6.12, stellt man Abweichungen fest. Da Maple vor Ausgabe der Modellfunktion mit den berechneten Parametern diese noch so weit wie möglich vereinfacht, sollten die Werte der Parameter nur den Ergebnissen unter “Show Results” entnommen werden.

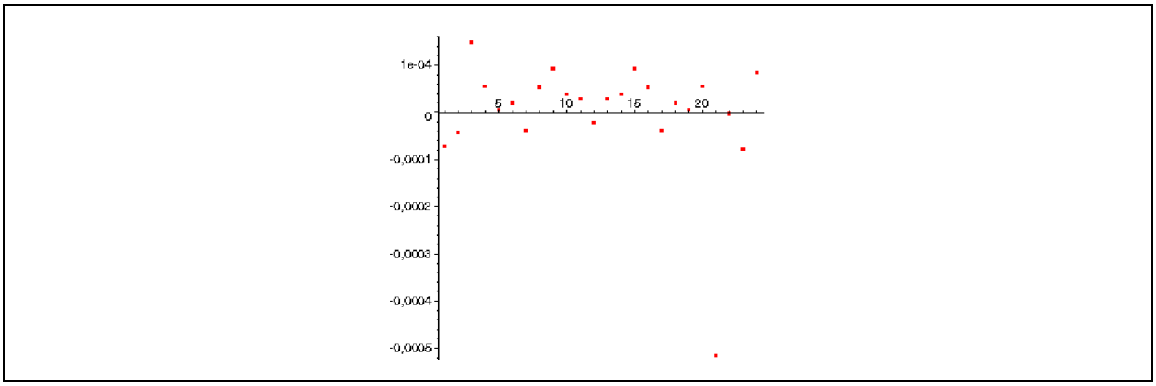


Abbildung 6.13: Graphische Ausgabe der Residuen der Funktion f nach der Ausgleichsrechnung

Abbildung 6.13 zeigt eine Darstellung der Residuen. Erkennbar ist eine gleichmäßige Streuung um die horizontale Achse, was für eine passende Modellfunktion spricht.

6.4.3 Angabe eines Parametergitters

Um die Startwerte für die Algorithmen festzulegen, bietet das Programm mehrere Möglichkeiten. Kann der Benutzer bereits eine gute Näherung für die Parameter vorgeben, können diese direkt eingegeben werden. Ist dies nicht der Fall, kann nach günstigen Startparametern über einem angegebenen Gitter gesucht werden. Für die Startwerte mit der geringsten Fehlerquadratsumme wird die Ausgleichsrechnung durchgeführt. Abbildung 6.14 zeigt die vom Programm bereitgestellte Eingabemaske. Die Anzahl der Parameter $a[i]$ ist in der graphischen Oberfläche auf zehn beschränkt. Diese Beschränkung bezieht sich nur auf die graphische Oberfläche, nicht aber auf die eigentlichen Algorithmen. Diese können mehr als zehn Parameter verarbeiten.

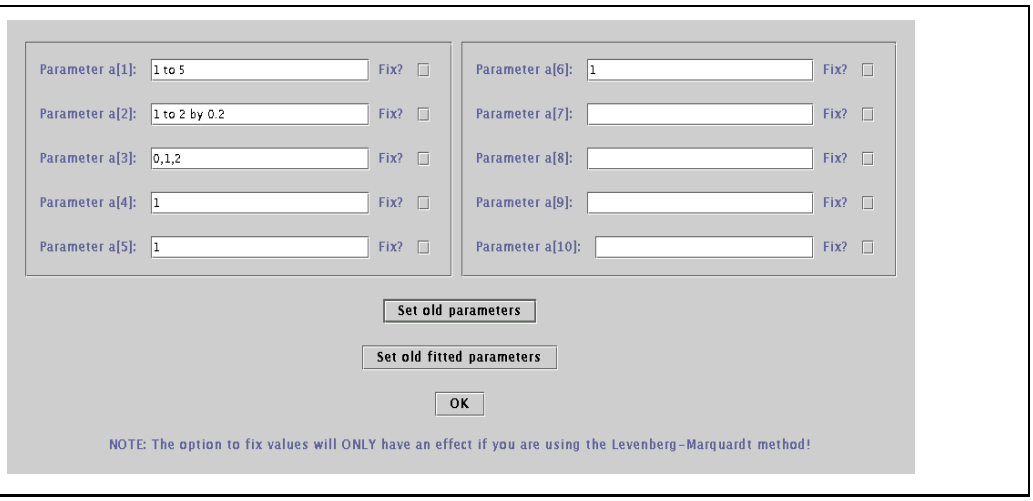


Abbildung 6.14: Eingabemaske für Startparameter

An der Beispieleingabe ist zu sehen, dass Parameter entweder direkt vorgegeben werden können oder aus einer Schleife berechnet werden. Die genaue Syntax zur Spezifikation der Parameterwerte findet sich in Abbildung 6.15. Über den Button “Fix?” kann der zugehörige Wert des Parameters während der Ausgleichsrechnung konstant gehalten werden. Über den Button “Set old fitted parameters” kann nach bereits erfolgter Ausgleichsrechnung mit den berechneten Parametern weiter iteriert werden. Das ist sinnvoll, falls die vorherige Berechnung z.B. aufgrund des Erreichens des Iterationsmaximums abgebrochen wurde.

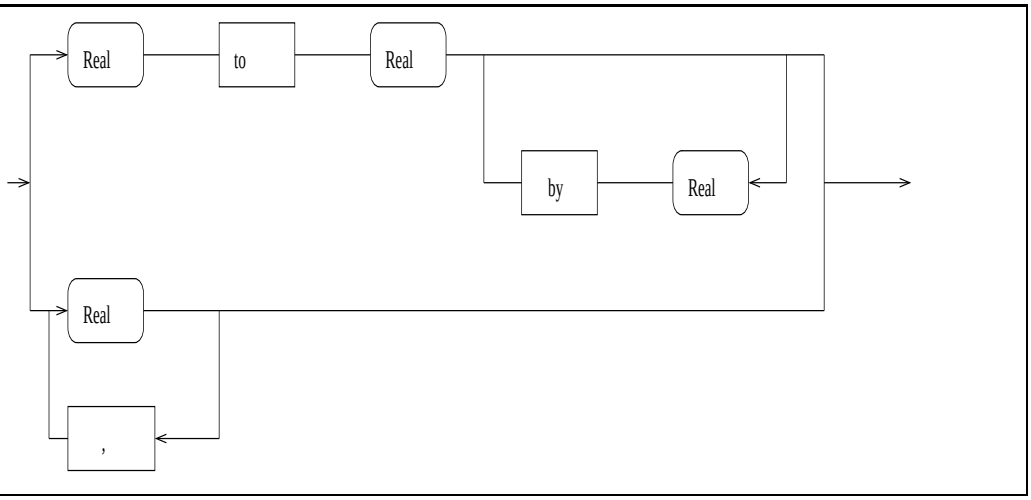


Abbildung 6.15: Syntaxdiagramm zur Angabe eines Startgitters

Kapitel 7

Zusammenfassung und Ausblick

In dieser Arbeit wurden Algorithmen zur Lösung nichtlinearer Ausgleichsprobleme sowie deren Implementierung in das Computeralgebrasystem Maple vorgestellt. Dabei wurden auf numerischer Seite der Levenberg-Marquardt- und ein modifizierter Gauß-Newton-Algorithmus eingesetzt.
Da die Genauigkeit der Ergebnisse durch Kenntnis der Ableitungen der betreffenden Modellfunktion verbessert wird, wurden Methoden beschrieben, diese symbolisch effizient zu berechnen.
In Bezug auf das Maple-System wurde eine Technik vorgestellt, um externe Programme in Quelltext oder aus einer Bibliothek in das Maple-System zu integrieren. Ebenso wurde ein Einblick in die programmierbare graphische Oberfläche (Maplets) des Maple-Systems gegeben, mit der das entstandene Programm versehen wurde.
Um den prinzipiellen Einsatz des Programmpakets sowohl unter der graphischen Benutzeroberfläche als auch im Worksheet zu demonstrieren, wurde dessen Funktionalität an einigen Beispielen erläutert. Ziel der abschließenden Untersuchungen war, die Vorteile und Schwächen der vorgestellten Algorithmen aufzuzeigen.

Das Tool wird derzeit noch weiterentwickelt. Langfristige Erweiterungen, die in Zukunft noch implementiert werden, sind dabei:

1. Hinzufügen weiterer Algorithmen zur Lösung von nichtlinearen Ausgleichsproblemen,
2. Einbeziehung von Nebenbedingungen,
3. Glättung der Kurven vor der Anpassung und automatische Erkennung und Entfernung von Ausreißern,
4. Erweiterung der graphischen Ausgabemöglichkeiten, wie z.B. die Angabe von Kurven, die Schranken darstellen, zwischen denen die gesuchte Funktion mit einer vorgegebenen Wahrscheinlichkeit verläuft,
5. Implementation von aufwändigen, in Maple geschriebenen Programmteilen, in C.

Der erste Punkt ist dabei angesichts der Resultate aus Kapitel 6 nicht zwingend erforderlich. Jedoch sollte man bedenken, dass die dort getesteten Datensätze nur einen repräsentativen Querschnitt darstellen. Das Hinzufügen weiterer Algorithmen kann sich also nur vorteilhaft auswirken.
Der fünfte Punkt bezieht sich insbesondere auf zeitintensive Routinen wie das Bestimmen von optimalen Startparametern. Da hier rekursiv vorgegangen wird, könnte eine Implementierung in C Vorteile bringen, falls das gewählte Parametergitter sehr groß ist.
Die Punkte zwei, drei und vier ergeben sich aus einem Vergleich mit einem ähnlichen Tool, der MATLAB Curve Fitting Toolbox ([22]). Dieses professionelle Programm bietet bereits die genannten Merkmale, besitzt allerdings im Gegensatz zu dem hier vorgestellten keine Möglichkeit, mehrdimensionale Ausgleichsprobleme zu behandeln.

Ein weiterer Punkt ist die Portierung des Programmpakets auf andere Betriebssysteme, wie z.B.

Windows. Aus dem Modulplan in Abbildung 6.4 ist ersichtlich, dass hierzu speziell das Modul `NonlinearFit_Linux` modifiziert werden muss. Die restlichen Module sind bereits systemunabhängig.
Ebenso sollte die Portierung auf zukünftige Maple-Versionen beachtet werden.

Anhang A

Verwendete Datensätze

Nr.	x[1]	x[2]	y	ω
1	-1	-0.5	-2.181405146	1
2	-1	0	0.248440510	1
3	-1	0.5	-0.857663991	1
4	-1	1.0	-2.366905869	1
5	-0.5	-1	3.788572344	1
6	-0.5	-0.5	-0.937964948	1
7	-0.5	0	-2.620906918	1
8	-0.5	0.5	0.287035520	1
9	-0.5	1.0	-3.669581410	1
10	0	-1	0.960930863	1
11	0	-0.5	0.248440510	1
12	0	0	-4	1
13	0	0.5	0.248440510	1
14	0	1.0	0.960930863	1
15	0.5	-1	-3.669581410	1
16	0.5	-0.5	0.287035520	1
17	0.5	0	-2.620906918	1
18	0.5	0.5	-0.937964948	1
19	0.5	1.0	3.788572344	1
20	1.0	-1	-2.366905869	1
21	1.0	-0.5	-0.857	1
22	1.0	0	0.2484	1
23	1.0	0.5	-2.1814	1
24	1.0	1.0	-1.2652	1

Tabelle A.1: Datensatz Fit2Dim

Nr.	x[1]	y	ω
1	1	494	1
2	2	759	0.25
3	3	1016	0.111111
4	4	1215	0.062500
5	5	1619	0.040
6	6	2964	0.027778
7	7	8489	0.020408
8	8	22377	0.015625
9	9	32618	0.012346
10	10	38153	0.01
11	11	41462	0.008264
12	12	42375	0.006944

Tabelle A.2: Datensatz Fit1Dim

Nr.	x[1]	x[2]	x[3]	y	ω
1	1	15	1	0.14	1
2	2	14	2	0.18	1
3	3	13	3	0.22	1
4	4	12	4	0.25	1
5	5	11	5	0.29	1
6	6	10	6	0.32	1
7	7	9	7	0.35	1
8	8	8	8	0.39	1
9	9	7	7	0.37	1
10	10	6	6	0.58	1
11	11	5	5	0.73	1
12	12	4	4	0.96	1
13	10	3	3	1.34	1
14	11	2	2	2.10	1
15	12	1	1	4.39	1

Tabelle A.3: Beispieldatensatz aus der NAG-Dokumentation

Anhang B

Beispiele für Eingabedateien

Das Format eines Datensatzes für das Programm hat folgendes Aussehen:

```
L
x_11 x_21 ... x_1L y_1 w_1
...  ...  ... ...  ... ...
x_N1 X_N2 ... x_NL y_N w_N
```

In der ersten Zeile steht die Anzahl der unabhängigen Variablen. In den weiteren N Zeilen stehen zunächst die L-dimensionalen Koordinaten des x-Vektors. Dahinter folgen die y-Koordinate sowie das Gewicht des Datenpunktes.
Im Folgenden werden die Datensätze A.2 und A.1 in dieser Form wiedergegeben.

Eindimensionaler Datensatz:

```
1
1     494   1
2     759   0.25
3    1016   0.111111
4    1215   0.062500
5    1619   0.040
6    2964   0.027778
7    8489   0.020408
8   22377   0.015625
9   32618   0.012346
10  38153   0.01
11  41462   0.008264
12  42375   0.006944
```

Zweidimensionaler Datensatz:

```
2
-1     -0.5   -2.181405146       1
-1     0      0.248440510        1
-1     0.5    -0.857663991       1
-1     1.0    -2.366905869       1
-0.5   -1     3.788572344        1
-0.5   -0.5   -0.937964948       1
-0.5   0      -2.620906918       1
-0.5   0.5    0.287035520        1
-0.5   1.0    -3.669581410       1
 0     -1     0.960930863        1
 0     -0.5   0.248440510        1
 0     0      -4                 1
 0     0.5    0.248440510        1
 0     1.0    0.960930863        1
0.5   -1     -3.669581410       1
0.5   -0.5   0.287035520        1
0.5   0      -2.620906918       1
0.5   0.5    -0.937964948       1
0.5   1.0    3.788572344        1
1.0   -1     -2.366905869       1
1.0   -0.5   -0.857             1
1.0   0      0.2484             1
1.0   0.5    -2.1814            1
1.0   1.0    -1.2652            1
```

Literaturverzeichnis

[1] R. FLETCHER: *Practical Methods of Optimization.* John Wiley, 2. ed. 2000

[2] P. E. GILL, W. MURRAY, M. H. WRIGHT: *Practical Optimization.* Academic Press, London, 1981

[3] A. L. PERESSINI, F. E. SULLIVAN, J. J. UHL: *The Mathematics of Nonlinear Programming.* Springer-Verlag, 1988

[4] D. G. LUENBERGER: *Linear and Nonlinear Programming.* Addison-Wesley Co., Inc., Reading, MA, 2. ed. 1984

[5] W. H. PRESS, S. A. TEUKOLSKY, W. T. VETTERLING, B. P. FLANNERY: *Numerical Recipes in C.* Cambridge Univ. Press, 2. ed. 2002
`http://www.library.cornell.edu/nr/bookcpdf.html`

[6] M. HANKE-BOURGEOIS: *Grundlagen der Numerischen Mathematik und des Wissenschaftlichen Rechnens.* Teubner Verlag, 1. Auflage 2002

[7] Statistical Reference Datasets
`http://www.itl.nist.gov/div898/strd/nls/nls_main.shtml`

[8] A. GRIEWANK: *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation.* SIAM Press 2002

[9] C. BISCHOF, M. BERZ, G. CORLISS, A. GRIEWANK: *Computational Differentiation.* SIAM Press, 1996

[10] J. GROTENDORST: *Computermathematik mit Maple.* Interner Bericht, ZAM, 2002, FZJ-ZAM-IB-2002-01

[11] D. VILLARD, M. B. MONAGAN: *Automatic differentiation: an implementation in Maple.* Report, 1998

[12] Introduction to Maplets
`http://www.mapleapps.com/categories/maple8/html/maplets.html`

[13] M. B. MONAGAN, K. O. GEDDES, K. M. HEAL, G. LABAHN, S. M. VORKOETTER, J. MCCARRON, P. DEMARCO: *Maple 8 Introductory Programming Guide.* Waterloo Maple Inc.

[14] M. B. MONAGAN, K. O. GEDDES, K. M. HEAL, G. LABAHN, S. M. VORKOETTER, J. MCCARRON, P. DEMARCO: *Maple 8 Programming Guide.* Waterloo Maple Inc.

[15] M. HÖRHAGER: *Maple in Technik und Wissenschaft.* Addison Wesley, 1996

[16] M. VAESSEN, R. KÜLHEIM: *Maple unter Unix*
Technische Kurzinformation, ZAM, 2003, FZJ-ZAM-TKI-0193
`http://www.fz-juelich.de/zam/docs/tki/tki_html/t0193/t0193.html`

[17] R. KÜLHEIM, J. GROTENDORST: *Solving Nonlinear Fitting Problems in Maple.* Poster Präsentation, Maple Summer Workshop 2002, 30.07.2002, University of Waterloo, Canada

[18] Online Dokumentation der NAG Fortran 90 Library
`http://www.nag.com/numeric/fn/fndescription.asp`

[19] A. KELLEY, I. POHL: *A Book on C.* Addison Wesley, 1999

[20] G. GROTEN: *Programmieren in Fortran 90/95.* Vorlesungsskript, FZJ-ZAM-BHB-0124

[21] Y. BARD: *Nonlinear Parameter Estimation.* Academic Press, 1974

[22] Dokumentation der MATLAB Curve Fitting Toolbox
`http://www.mathworks.com/access/helpdesk/help/toolbox/curvefit.shtml`